COMPTE RENDU DES TRAVAUX PRATIQUES 1, 2, 3, 4 EN STRUCTURES DES DONNÉES AVANCÉES

Travail réalisé par:

YABA BILONGO Michel Davel
MACHOUCHE Kahina

Introduction

Les structures de données permettent de faire des opérations sur les ensembles dynamiques. Dans ce cours, plusieurs structures de données ont été abordé parmi lesquelles les tables dynamiques, les tas binaires, les tas binomiaux, les b-arbres et les AVL. Dans le cadre de ces travaux pratiques, il est question de faire une analyse amortie dans chacune de ces structures de données.
De ce fait, nous allons d'abord faire une analyse des opérations d'insertions dans une table dynamique,. En second lieu, nous allons faire une analyse dans le cas où on effectue à la fois des insertions et suppressions dans une table dynamique. En troisième lieu, nous présenterons l'analyse dans un les tas binaires et binomiaux. Et enfin, nous ferons une analyse dans un B-Arbre et dans les AVL.

Les détails des implémentations et des expériences se trouvent sur le compte Github dont le lien est :

https://github.com/MichelYABA/TPS-SDA.git

TP 1: INSERTIONS DANS UNE TABLE DYNAMIQUE

1) Définition de la fonction du potentiel

Les opérations dans une structure de données ont une complexité en temps qui souvent dépend de l'état courant de la structure de données, ce qui rend l'analyse d'une suite de n opérations souvent délicate.
Pour faire une analyse amortie du coût réel d'une séquence de n opérations, on utilise plusieurs techniques parmi lesquelles celle de la méthode du potentiel.
La méthode du potentiel considère une fonction du potentiel notée Φ qui amortit le coût de l'extension.
Généralement quand seules les insertions sont effectuées dans une table pleine, on double la taille du tableau. La fonction du potentiel correspondant est alors $\Phi = 2n(T) - taille(T)$
avec n(T) : nombre d'éléments dans la table T et taille(T) : taille du tableau T.

Soit $\Phi = \frac{2n(T) - taille(T)}{2-1} = \frac{2n(T) - taille(T)}{1} = 2n(T) - taille(T)$

Donc quand on décide de multiplier la taille du tableau par un facteur $\alpha \geq 1$
on a :

$\Phi = \frac{\alpha n(T) - taille(T)}{\alpha - 1}$

2/ Coût amorti de l'opération « insérer table » en fonction de α

Par définition : $\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$

Avec :

- c_i : le coût réel de la $i^{ième}$ opération
- $\hat{c}_i$: le coût amorti de la $i^{ième}$ opération
- Φ_i : fonction du potentiel avant l'opération
- Φ_{i-1} : fonction potentielle après l'opération

Soit :

- $t(i-1) = t_{i-1}$: la taille du tableau avant l'insertion
- $t(i) = t_i$: la taille du tableau après l'insertion
- $n(i-1) = n_{i-1}$: le nombre d'éléments avant l'insertion
- $n(i) = n_i$: le nombre d'éléments après l'insertion

cas 1 : L' ajout du $i^{ième}$ élément ne permet pas d'étendre la table (la table n'est pas pleine)

Dans ce cas, on a : $c_i = 1$, $t_{i-1} = t_i$, $n_{i-1} = n_i - 1$

$$\hat{c}_i = 1 + \frac{\alpha n(i) - t(i)}{\alpha - 1} - \frac{\alpha n(i-1) - t(i-1)}{\alpha - 1}$$

$$= 1 + \frac{\alpha n(i) - t(i) - \alpha n(i) + \alpha + t(i)}{\alpha - 1}$$

$$\hat{c}_i = 1 + \frac{\alpha}{\alpha - 1}$$

cas 2 : L'ajout d'un élément permet l'extension de la table

Dans ce cas, on a : $c_i = n_i$, $n_{i-1} = n_i - 1$, $t_i = \alpha\, n_i$ (ceci est donné comme énoncé)

On ne connaît pas la taille du tableau.
Or la taille du tableau après c'est la taille du tableau avant multiplié par un facteur α, donc $t_{i-1} = n_i$

$$\hat{c}_i = n_i + \frac{\alpha n(i) - t(i)}{\alpha - 1} - \frac{\alpha n(i-1) - t(i-1)}{\alpha - 1}$$

$$= n_i + \frac{\alpha n(i) - \alpha n(i)}{\alpha - 1} - \frac{\alpha(n(i) - 1) - n(i)}{\alpha - 1}$$

$$\hat{c}_i = \frac{\alpha}{\alpha - 1}$$

3/

a) **<u>Morceau plus lent</u>**

Le morceau de code le plus lent est la boucle de la figure ci-dessous :

```
for(i = 0; i < 1000000 ; i++){
    // Ajout d'un élément et mesure du temps pris par l'opération.
    before = System.nanoTime();
    memory_allocation = a.append(i);
    after = System.nanoTime();

    // Enregistrement du temps pris par l'opération
    time_analysis.append(after - before);
    // Enregistrement du nombre de copies efféctuées par l'opération.
    // S'il y a eu réallocation de mémoire, il a fallu recopier tout le tableau.
    copy_analysis.append( (memory_allocation == true)? i: 1);
    // Enregistrement de l'espace mémoire non-utilisé.
    memory_analysis.append( a.capacity() - a.size() );
}
```

Complexité des fonctions de ce morceau de code

Les fonctions de ce morceau de code dont on doit mesurer la complexité sont les fonctions «append » qui consiste à ajouter les éléments dans la table dynamique.
Si la table courante n'est pas pleine, alors la complexité d'une opération d'insertion est de O(1) puisqu'on n'a besoin d'effectuer qu'une seule insertion élémentaire. En revanche, si la table courante est pleine et qu'une extension a lieu, alors le coût réel de la $i^{ème}$ opération est O(i). En effet, le coût est 1 pour l'insertion élémentaire plus i−1 pour les éléments qui doivent être copiés à partir de l'ancienne table vers la nouvelle.

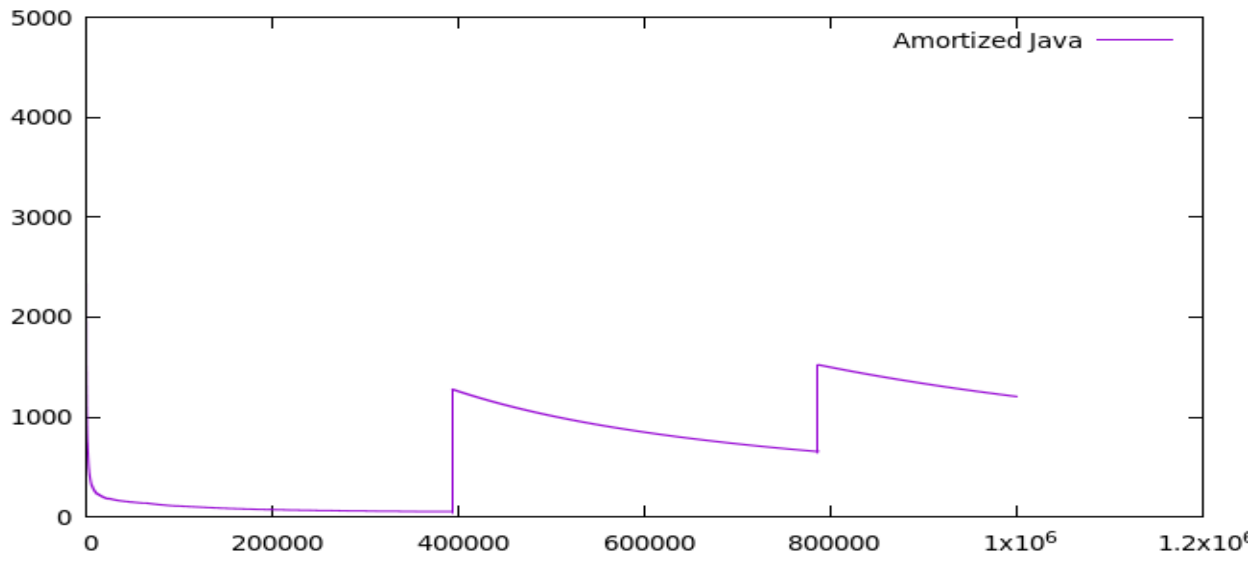
Comme n opérations sont effectuées, le coût le plus défavorable d'une opération est O(n), ce qui conduit à un majorant de $O(n^2)$ pour le temps d'exécution total de n opérations.

Le Pourquoi du fait que c'est le morceau de code le plus lent

Ce morceau de code est le plus lent parce qu'il doit être exécuté autant de fois qu'il y a des valeurs à ajouter dans la table dynamique.

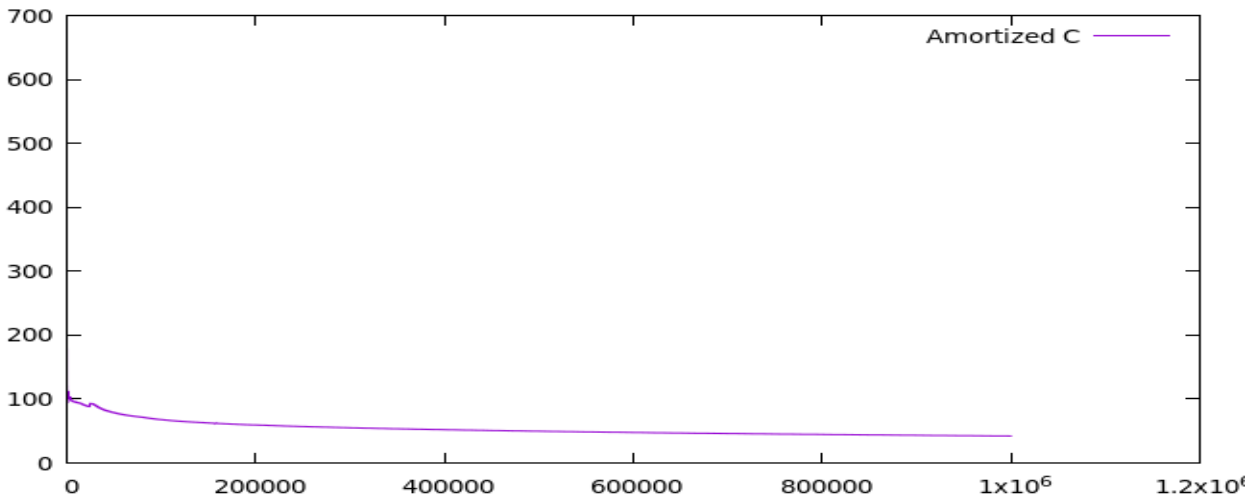
b) **Observations des coûts amortis dans différents langages**

Coût amorti avec Java



En observant cette courbe, constate que le coût amorti augmente au début, au milieu et vers la fin des opérations.
Le coût amorti augmente pour compenser la sous-facturation des opérations ultérieures.

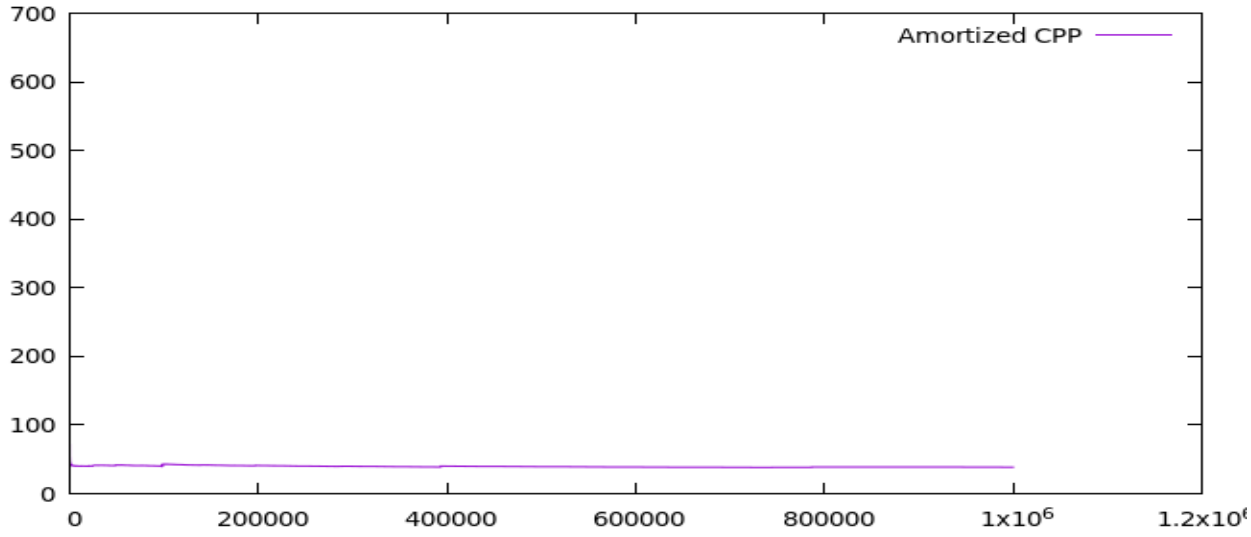
Coût amorti en C



On constate que le coût amorti augmente uniquement en début.

Le coût amorti augmente en début pour compenser la sous-facturation des opérations ultérieures.

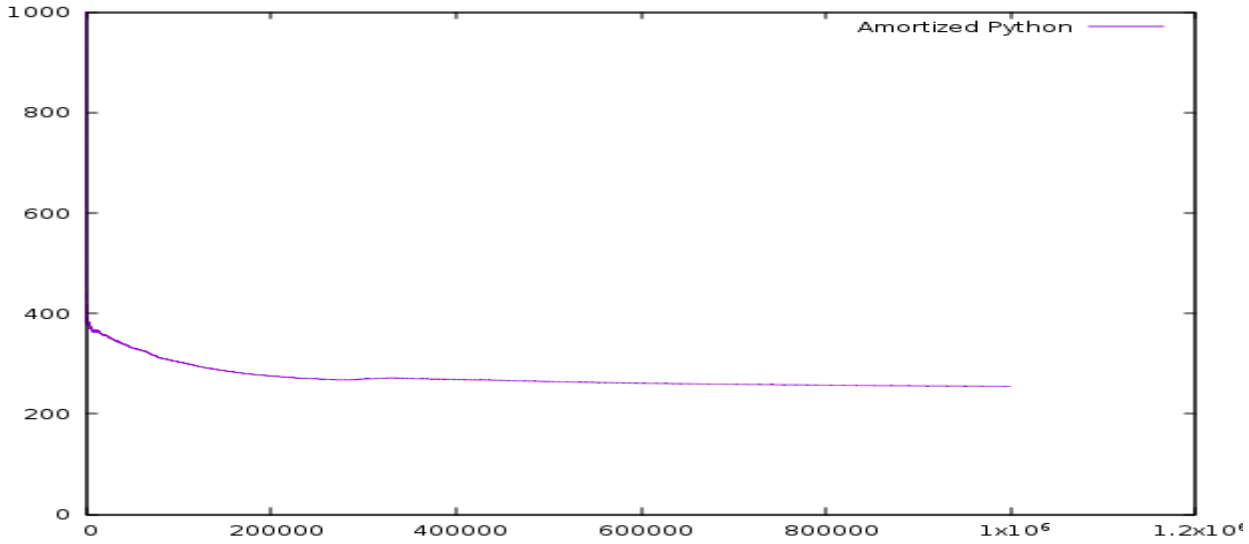
Coût amorti en C++



On constate que le coût amorti augmente uniquement en début.

Le coût amorti augmente en début pour compenser la sous-facturation des opérations ultérieures.

Coût amorti en Python

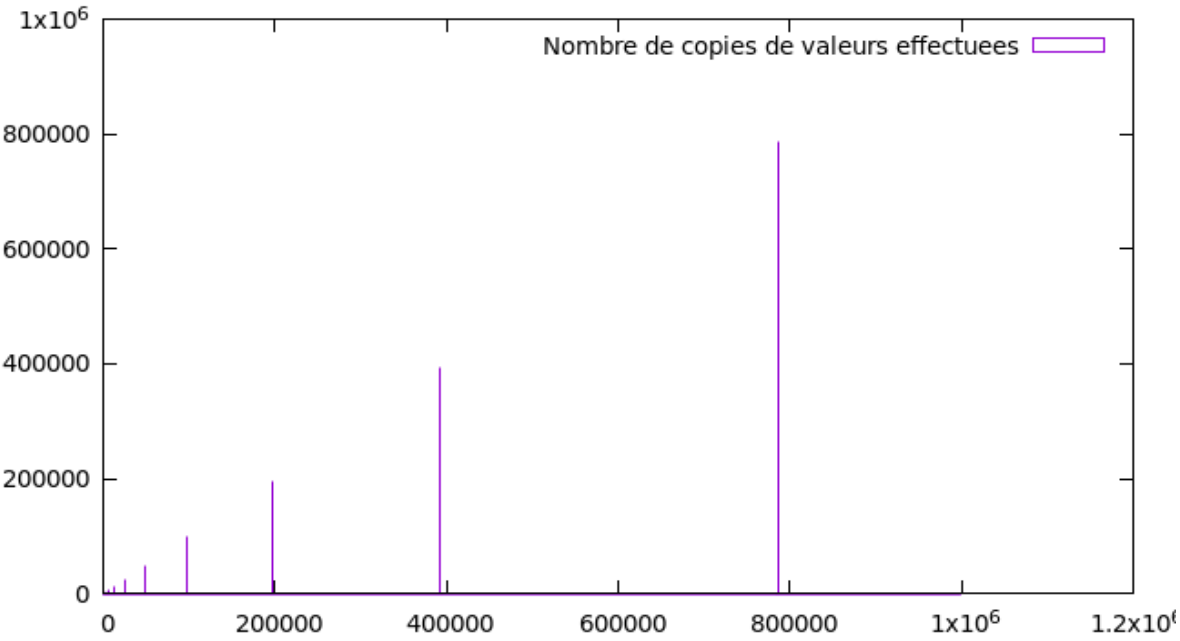


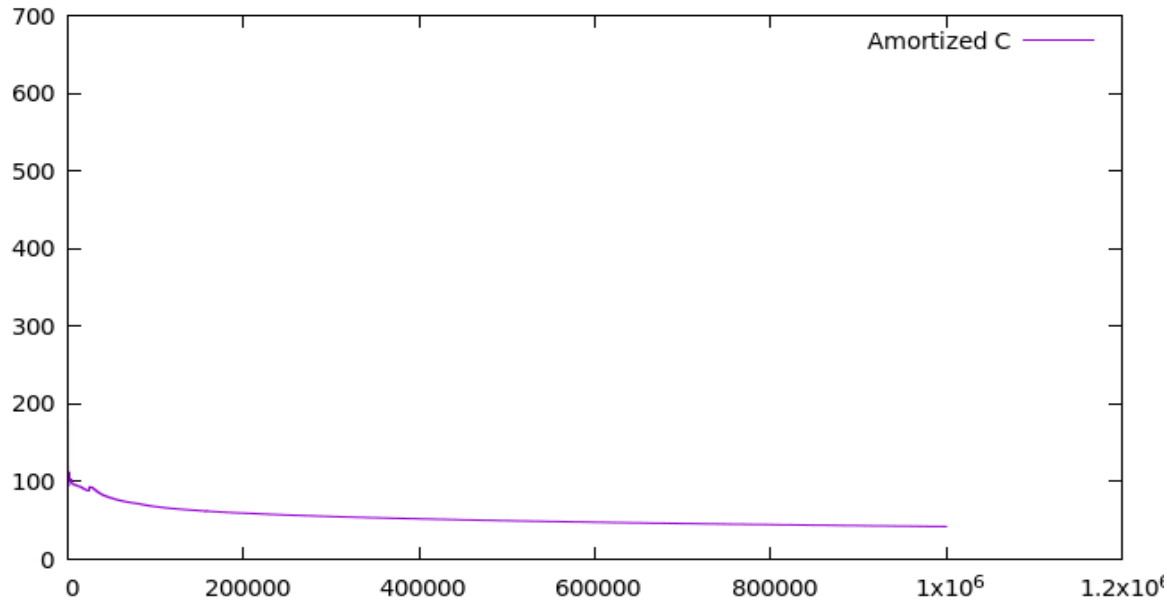
On constate que le coût amorti augmente uniquement en début.

Le coût amorti augmente en début pour compenser la sous-facturation des opérations ultérieures.

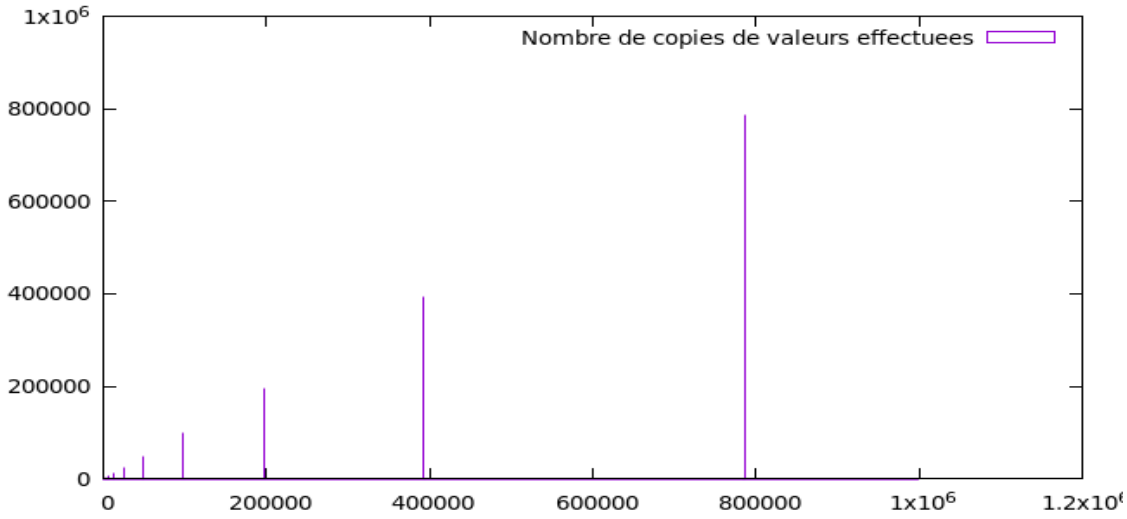
c) Nombre de copies effectuées et coûts amortis

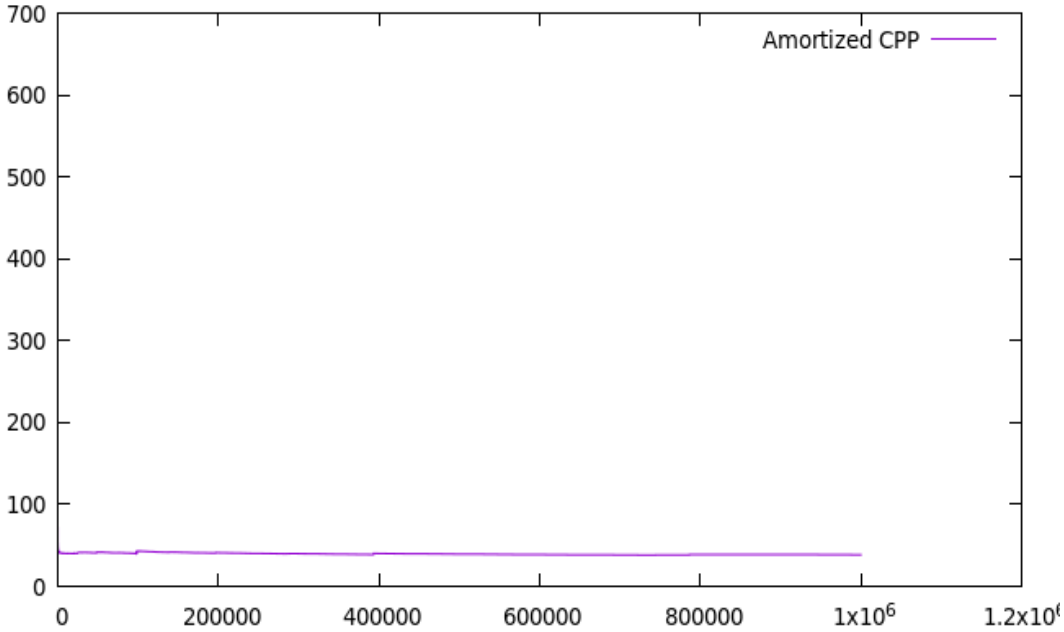
En C



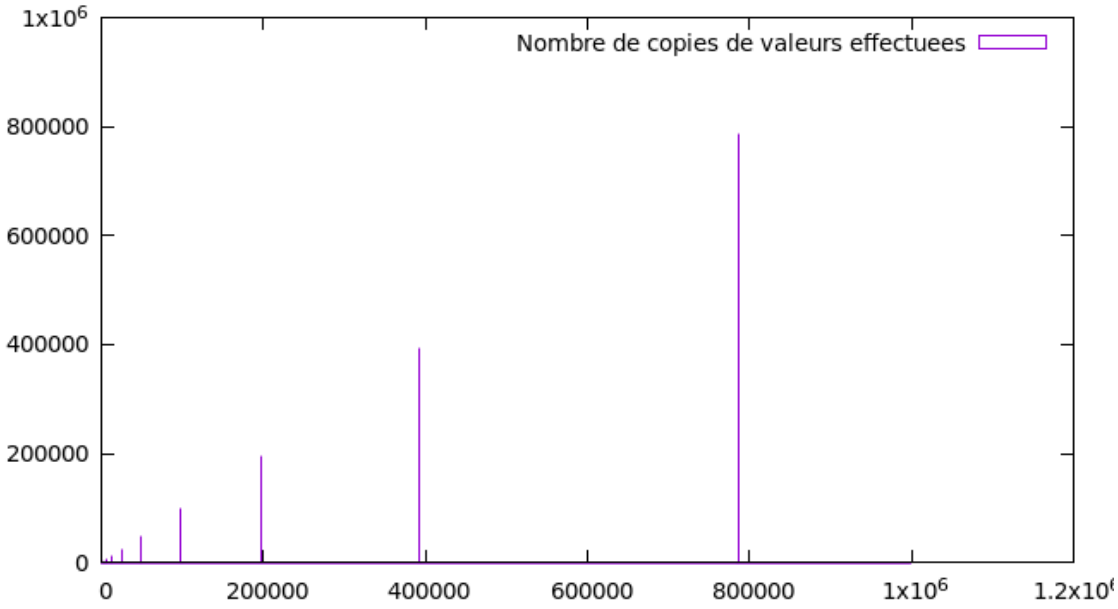

Amortized C
700
600
500
400
300
200
100
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

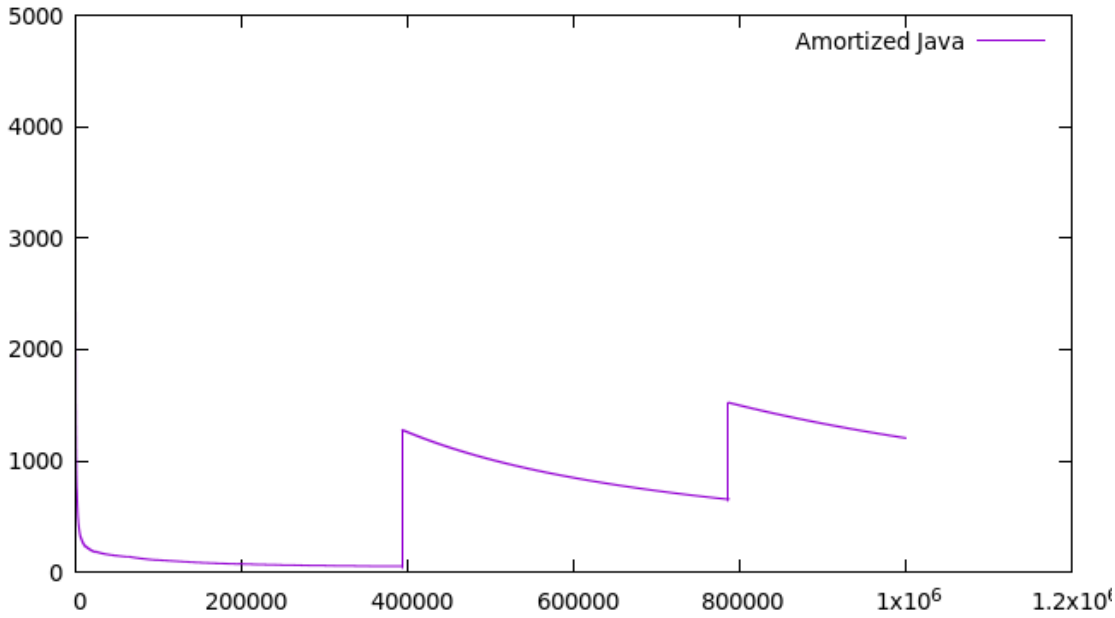
En C++


Nombre de copies de valeurs effectuees
1x10^6
800000
600000
400000
200000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6


Amortized CPP
700
600
500
400
300
200
100
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

En Java





Conclusion sur le nombre de copies

Dans chaque langage, le nombre de copies augmente d'une part et reste au plus bas de l'autre coté. Il y a augmentation de nombre de copies chaque fois que la table est remplie au 3/4 de sa capacité. Cette augmentation du nombre de copies est telle qu'il y a autant de copies qu'il y a d'éléments dans la table (Par exemple, pour n ~ 200000, on a ~ 200000 copies, autant pour n~400000, n~800000, etc). L'augmentation du nombre de copies est exponentielle.

Conclusion sur le coût amorti

En Java, le coût amorti augmente quand le nombre de copies augmente également.
En C et C++, le coût amorti augmente au début quand il y a moins de copies et diminue quand le nombre de copies augmente.

Différence avec le temps réel
Le coût amorti dépend du coût réel

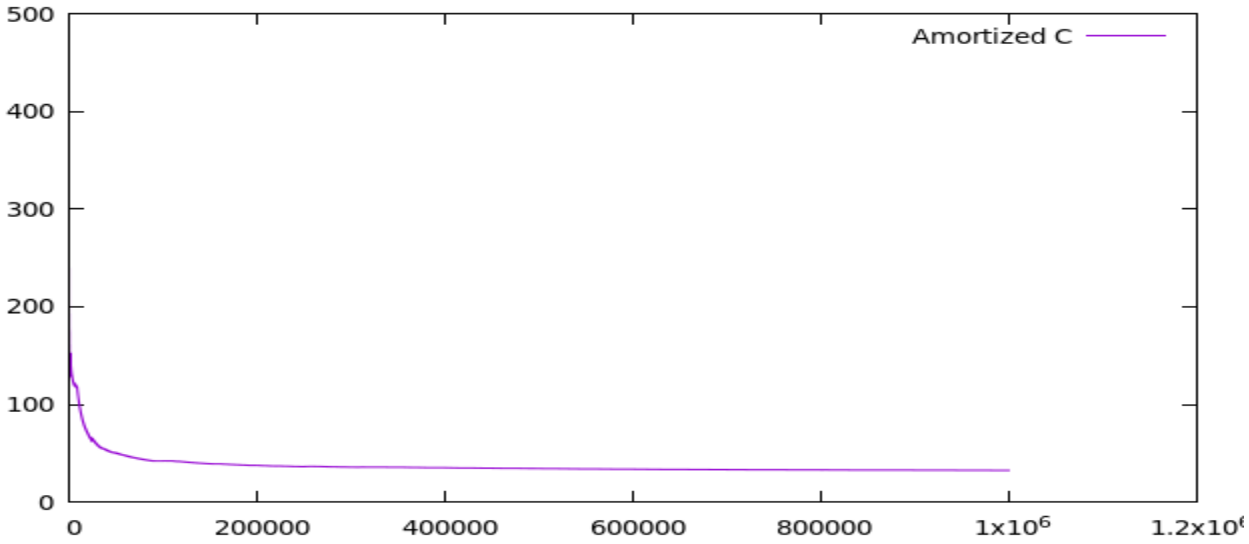
n'y a pas de réelle différence entre le coût réel et le nombre de copies car le coût réel dépend du nombre de copies effectuées . S'il y a extension de la table, le coût réel correspond au nombre de copies des valeurs de l'ancienne table dans la nouvelle augmentée par la valeur qu'on veut ajouter. Par contre s'il n'y a pas d'extension de table, le nombre de copie est égale au coût réel c'est-à-dire 1.

d) Expériences dans différents langages :

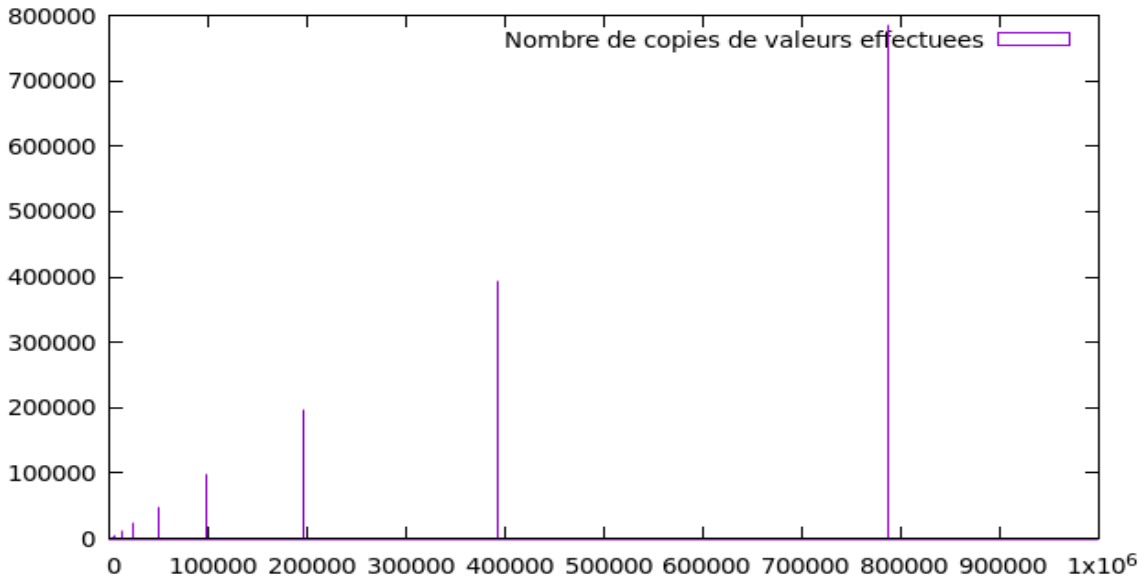
En C :

Expérience 1 :

Coût amorti

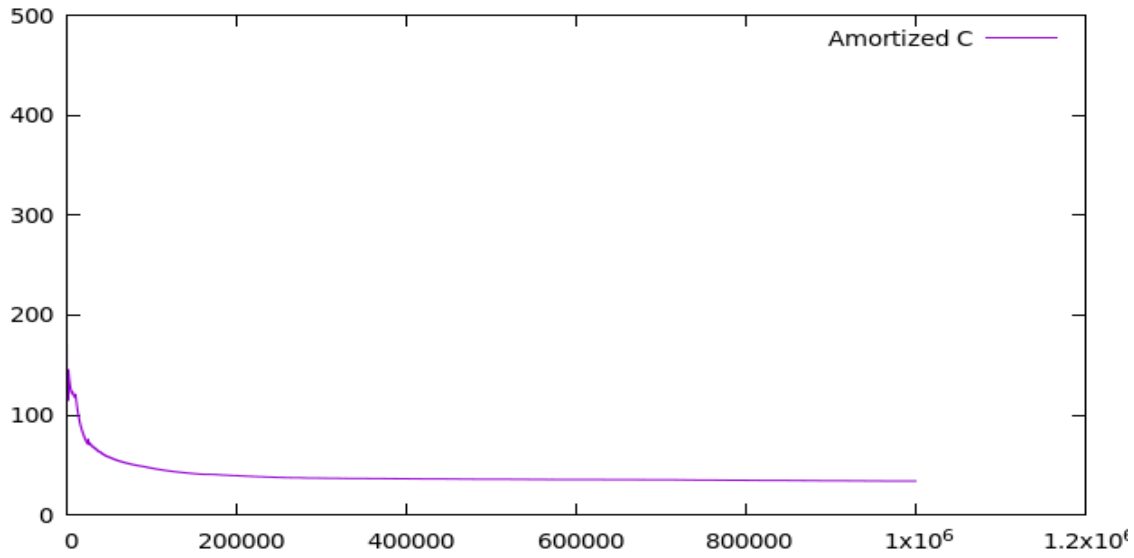


Nombre de copies :

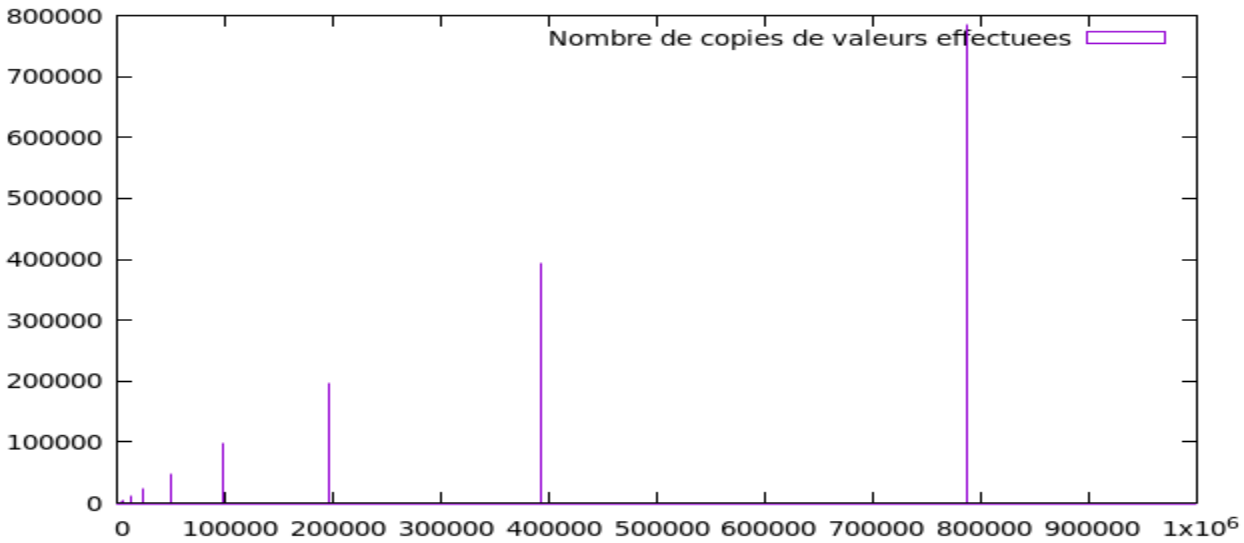


Expérience 2 :

Coût amorti

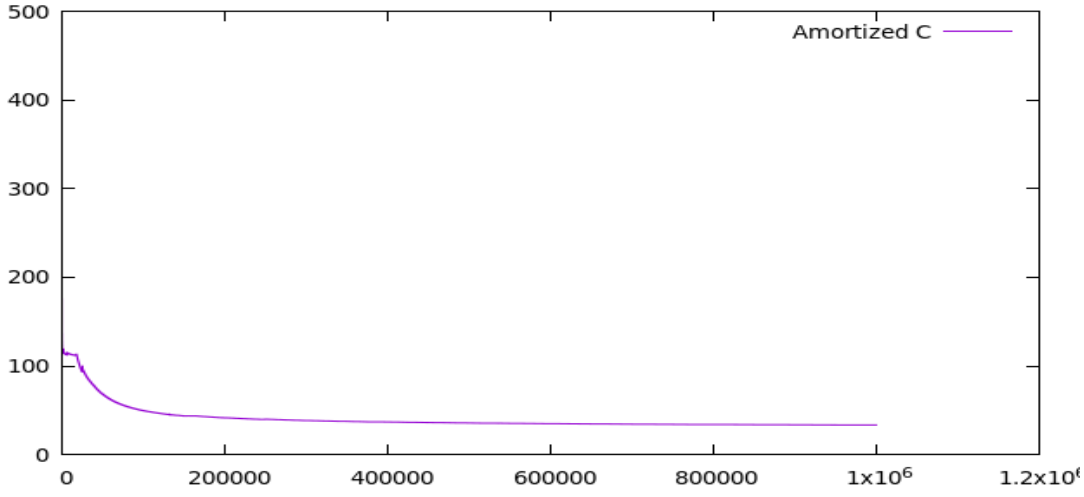


Nombre de copies :

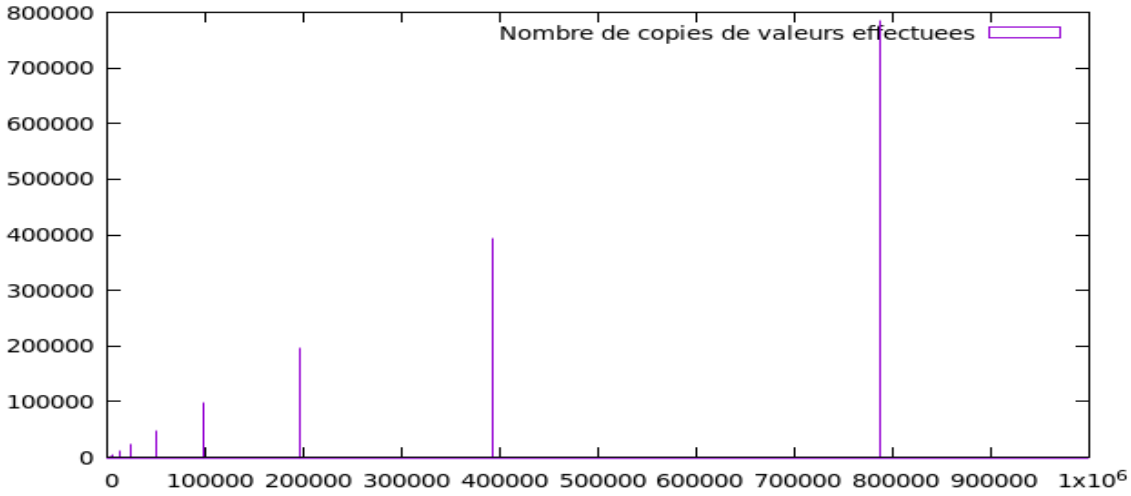


Expérience 3 :

Coût amorti



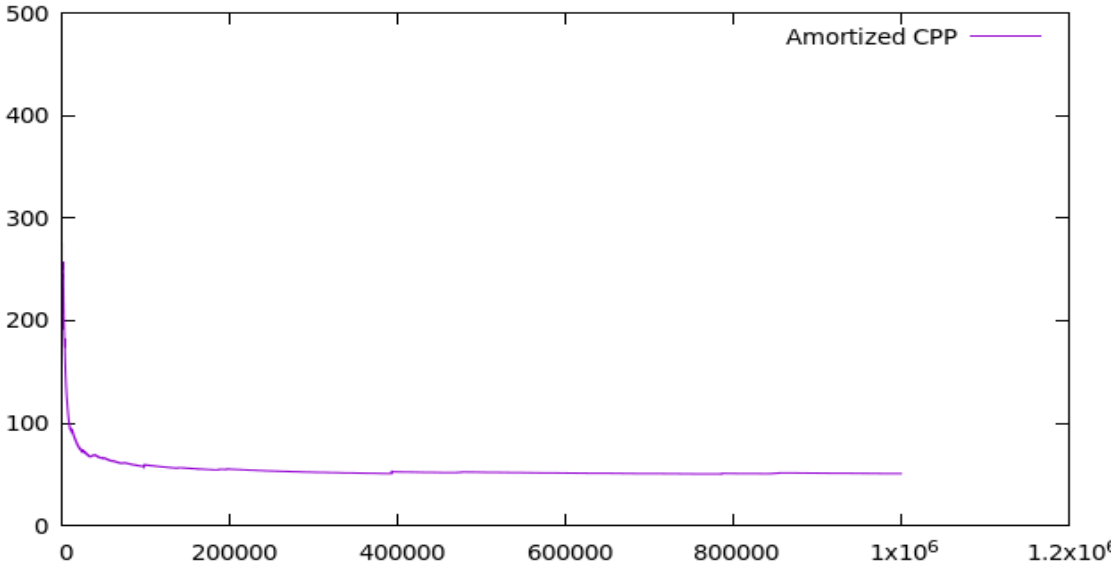
Nombre de copies :



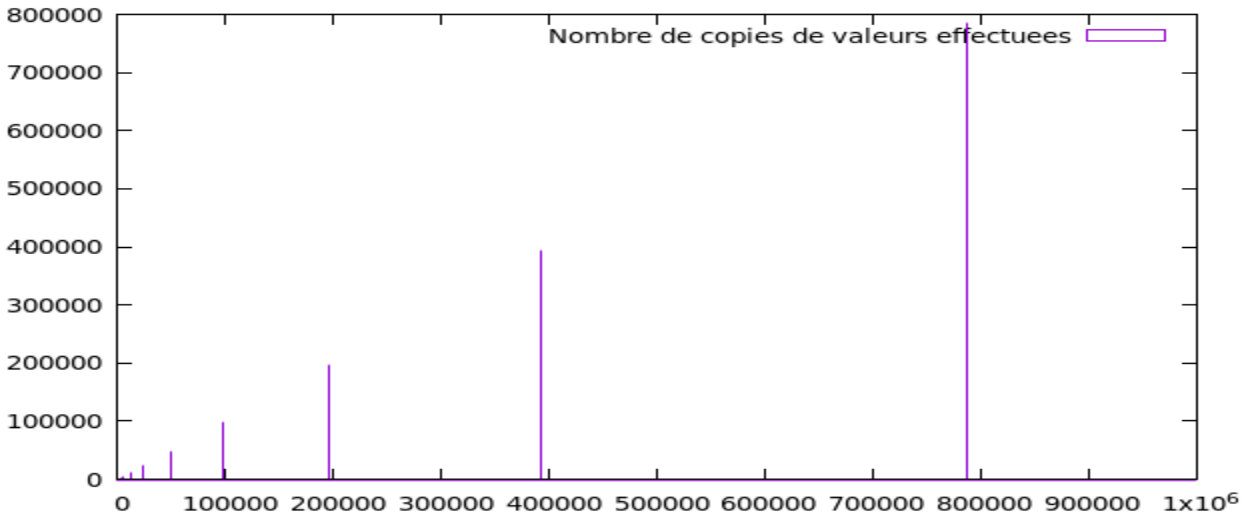
En C++

Expérience 1

Coût amorti :

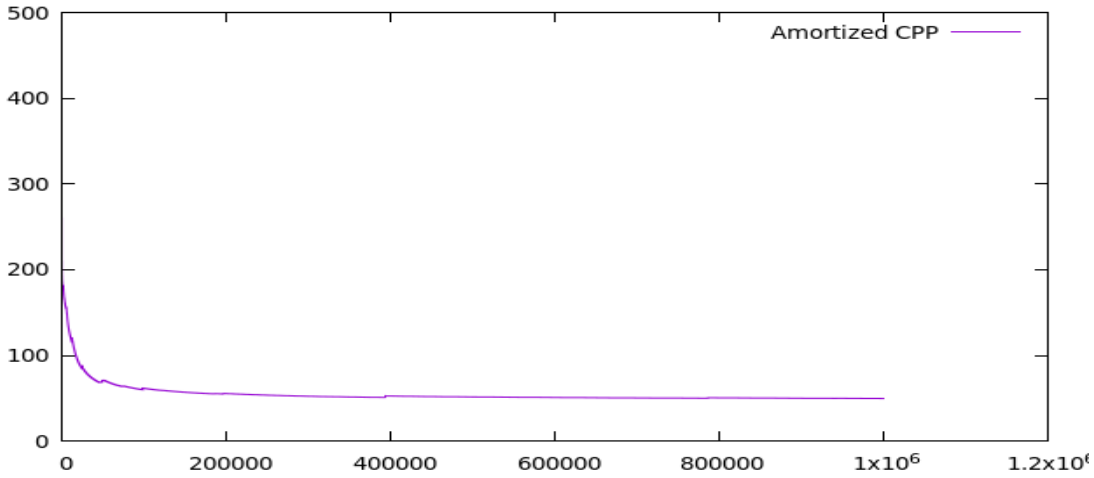


Nombre de copies :

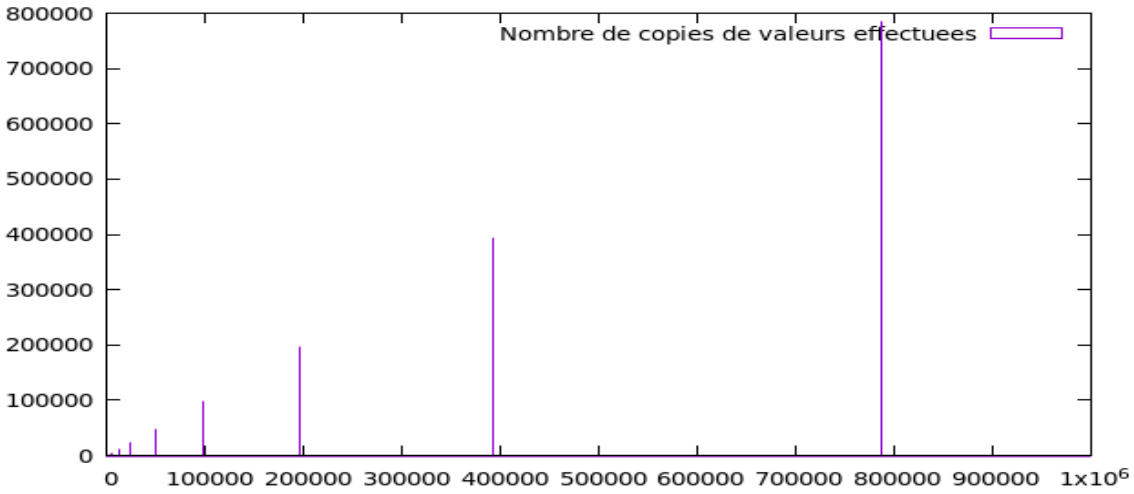


Expérience 2

Coût amorti :

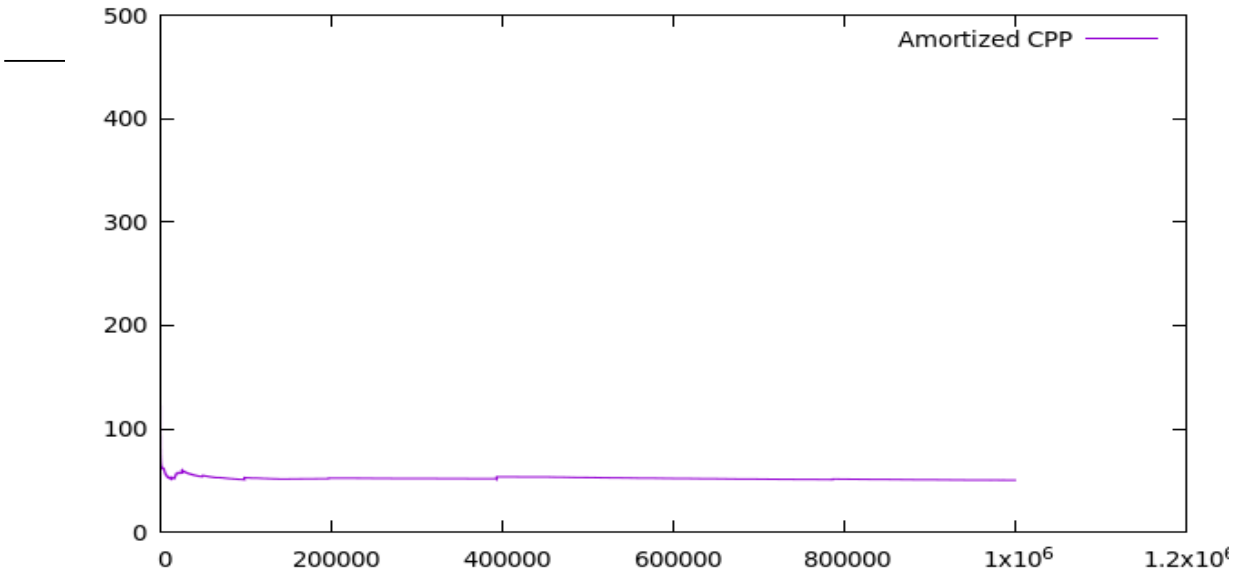


Nombre de copies :

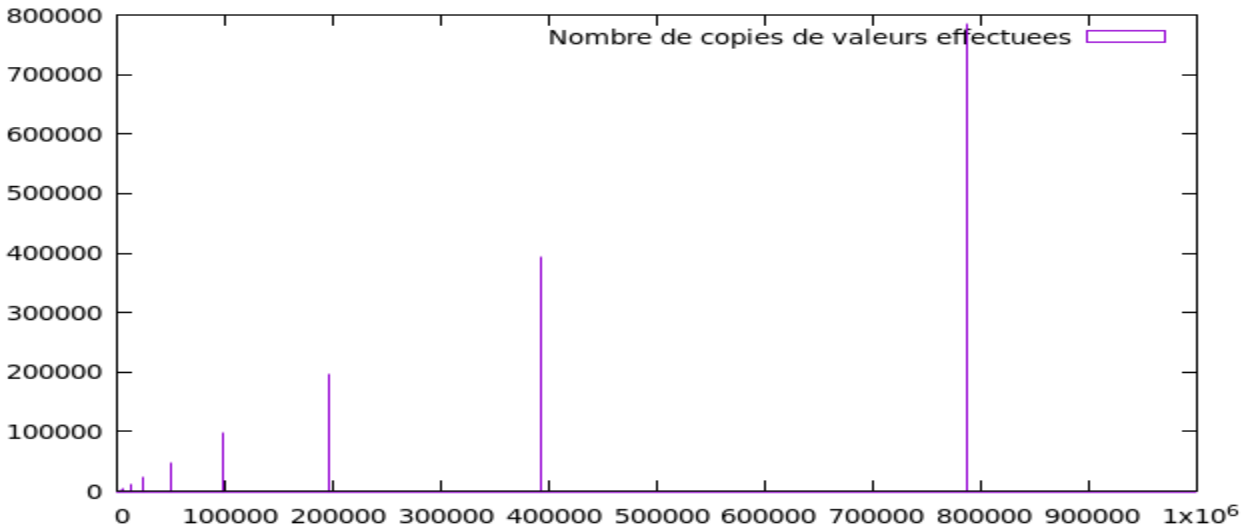


Expérience 3

Coût amorti :



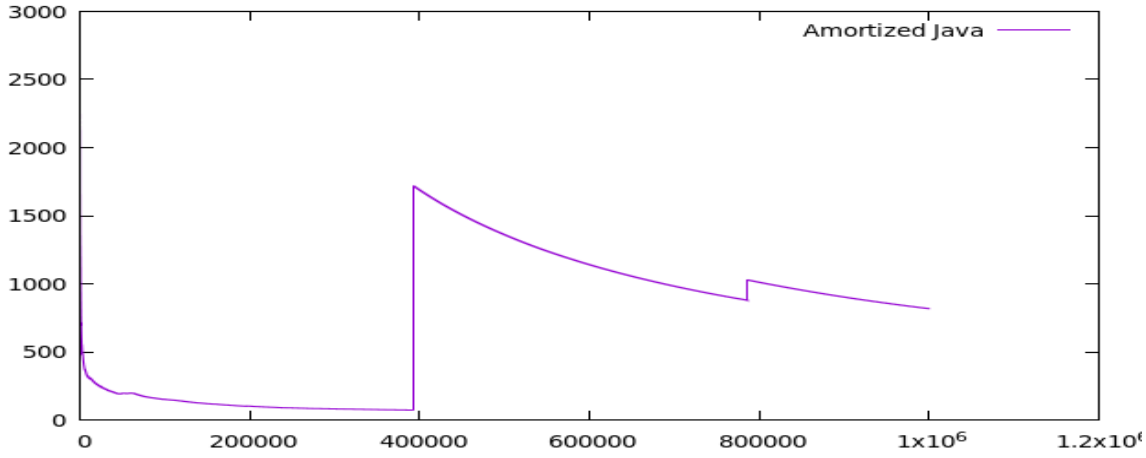
Nombre de copies :



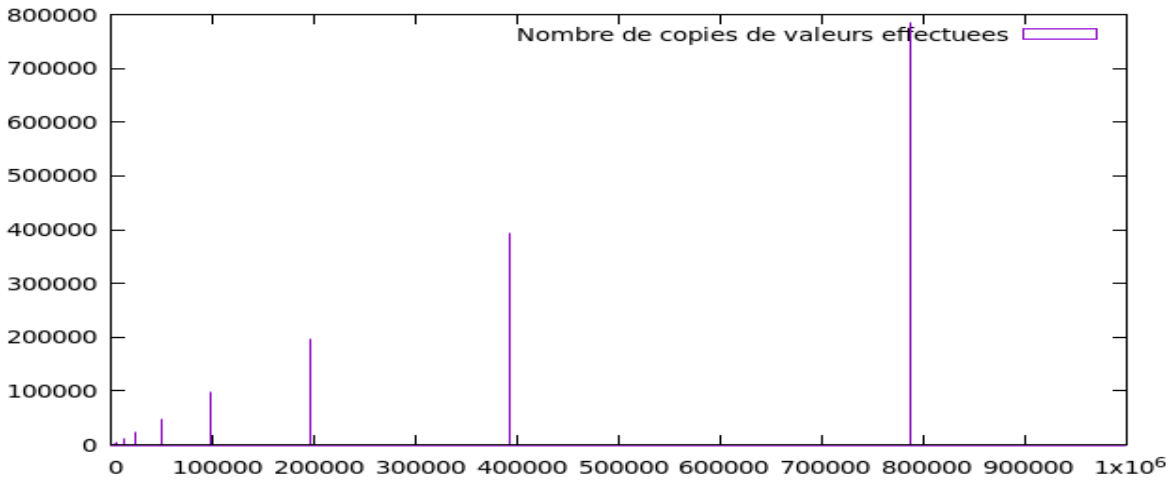
En Java

Expérience 1

Coût amorti :

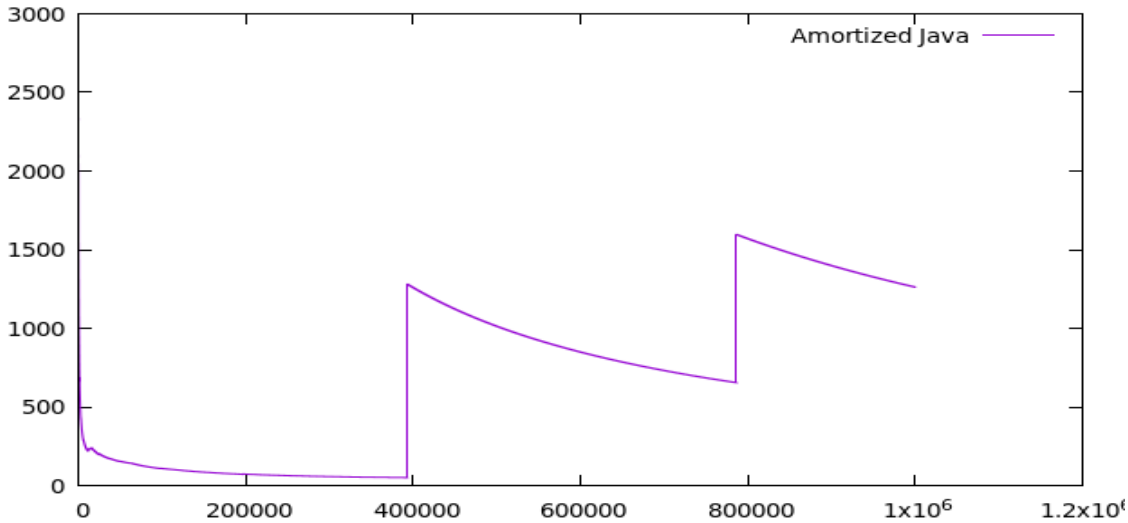


Nombre de copies :

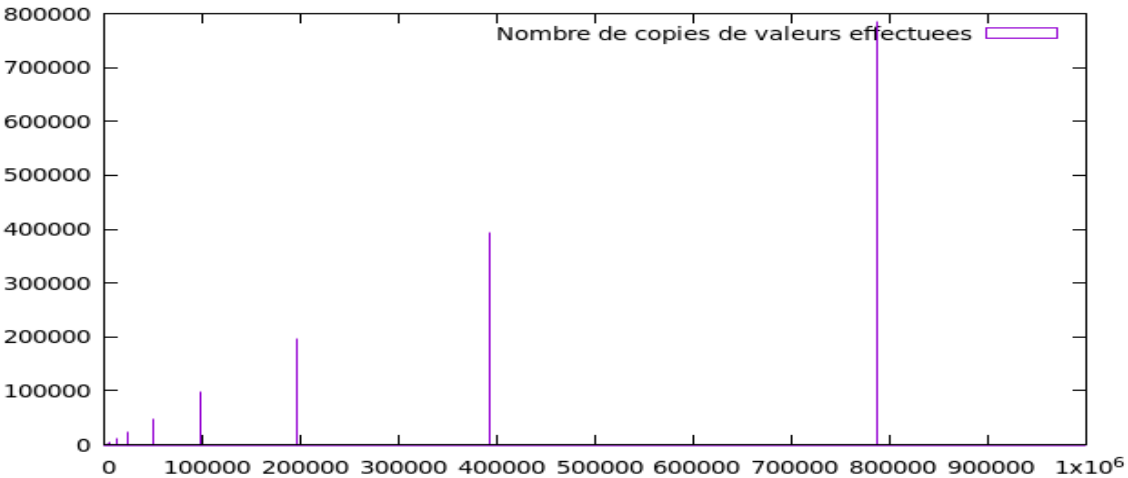


Expérience 2

Coût amorti :

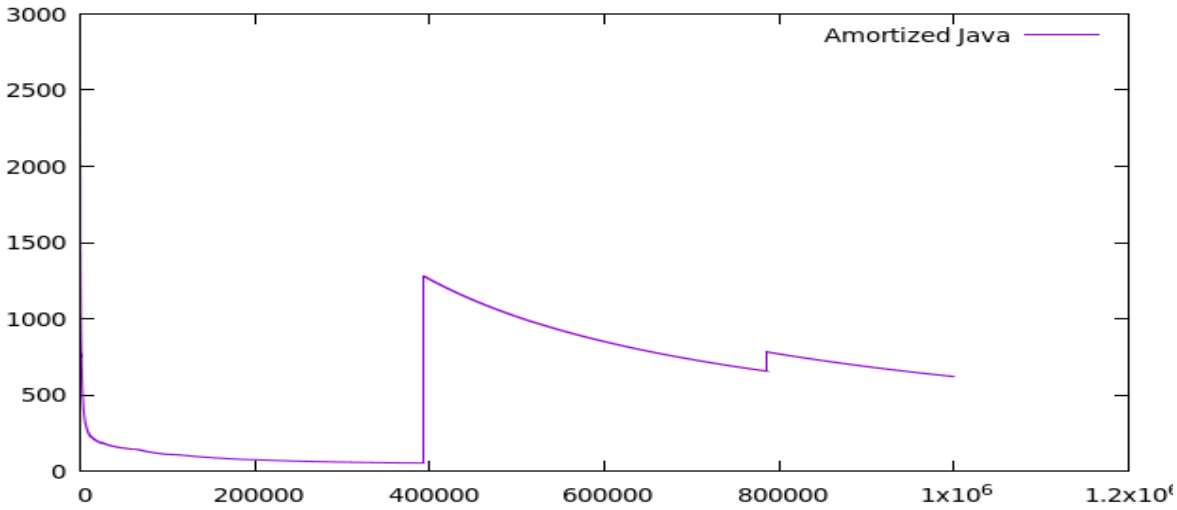


Nombre de copies :

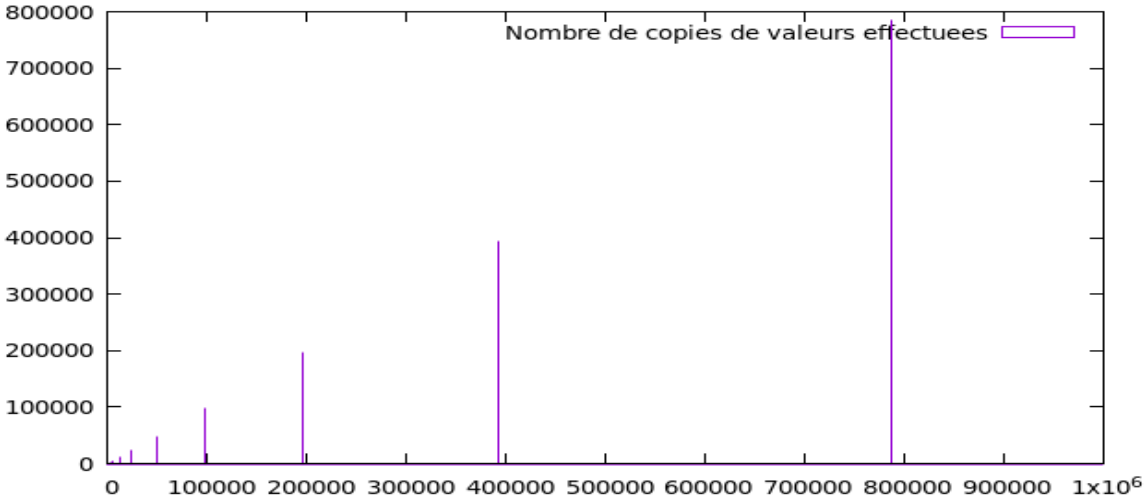


Expérience 3

Coût amorti :



Nombre de copies :



<u>Conclusion :</u>

En recommençant l'expérience plusieurs fois avec différents langages (C/C++/Java), nous constatons que :

- Les coûts réels et amortis changent d'une expérience à une autre
- Le nombre de copies ainsi que la mémoire inutilisée reste inchangée d'une expérience à une autre

<u>Justification</u> :

Parce que d'autres processus s'exécutent en même temps que nos programmes, voilà pourquoi les coûts varient d'une expérience à une autre.

e) Explication de pourquoi certains langages sont plus rapides que d'autres dans cette expérience

En exécutant le programme dans différents langages, on constate, à travers les coûts générés dans chacun, que certains langages sont plus rapides que d'autres.

En effet, les coûts des exécutions dans chaque langage nous donne ceci :

En C :

```
Total cost: 32420367.000000
Average cost: 32.420367
Variance: 16601955105.919804
Standard deviation: 128848.574326
```

En C++ :

```
Total cost :5.06436e+07
Average cost :50.6436
Variance :7.32672e+11
Standard deviation :855962
```

En Java :

En

```
Total cost : 1262464500
Average cost : 1262.4645
Variance :782711631022960065.38623975
Standard deviation :884709913.4874436855316162109375
```

Python :

```
Total cost : 342312335.9680176
Average cost : 342.3123359680176
Variance :195835124176.59845
Standard deviation :442532.62498554663
```

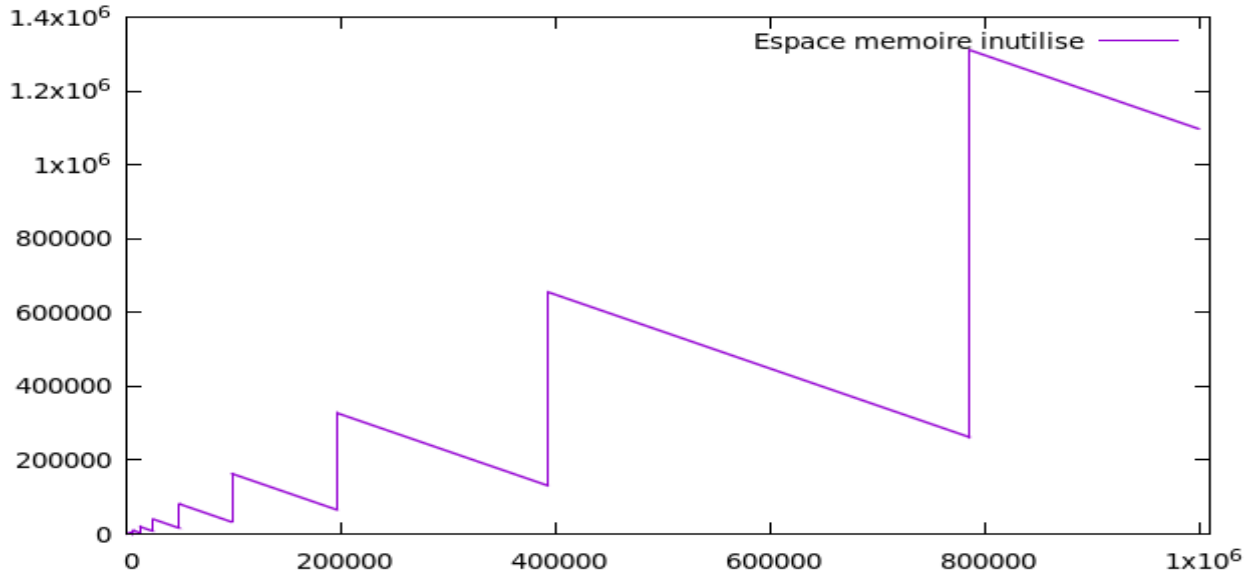
Nous voyons clairement que C est plus rapide que C++, qui à son tour est plus rapide que Python, ce dernier étant plus rapide que Java.

Cette différence s'explique par le type de langage. En effet, C/C++ sont des langages compilés et Java/Python sont des langages interprétés.

- Les langages interprétés sont des langages dans lesquels le code source est interprété par un un logiciel appelé « interpréteur ». Il revient à l'interpréteur d'exécuter les lignes de code une par une et il va décider de ce qu'il va faire par la suite. L'un de ses avantages c'est que le même code source peut marcher directement sur tout ordinateur. Néanmoins, Parce que les programmes ne sont pas directement exécutés par le système d'exploitation, mais par un interpréteur, voilà pourquoi les programmes écrits dans ces langages prennent plus de temps d'exécution
- Les langages compilés sont des langages dont le code source est d'abord compilé, par un logiciel appelé « compilateur », en code binaire qui est très facile à lire pour un ordinateur. C'est alors directement le système d'exploitation qui va utiliser le code binaire pour exécuter le programme. De ce fait, les programmes écrits dans ces langages sont plus rapides que ceux écrits en langages interprétés. Néanmoins, il faudra (en général) tout recompiler quand on passe d'un ordinateur à un autre.

Si Java accélère dans certains cas, c'est qu'il compile les bouts de code réutilisés.
En C++, Java et Python, on a réécrit une classe qui en appelle une autre, qui fait les mêmes tests.

f) **Mémoire inutilisée**



Analyse

Nous constatons que :

- L'espace mémoire inutilisée est d'autant plus grand qu'il y a d'éléments dans la table
- L'espace mémoire inutilisée augmente quand la table est rempli au 3/4 de sa capacité
- l'augmentation de l'espace mémoire inutilisé peut dépasser le nombre d'éléments qu'il faut dans la table
- Après avoir ajouté 10^6 nombres, il reste plus un nombre supérieur à 10^6 de cases non utilisés

Ce qu'on en pense

Suite à notre analyse, on pense qu'il y a énormément de gaspillage de mémoire.
Ce qui n'est pas optimale car étendre la table avant même qu'elle ne soit pleine implique un coût total plus important.
Une solution est plutôt d'étendre la table que lorsqu'elle est pleine et ce au fur et à mesure qu'on ajoute d'élément dans la table.

Exemple de scénario

On veut ajouter les nombres 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23 dans une table de capacité initiale 4. Soit n=4
soit la table avec 4 alvéoles :

On ajoute le nombre 0 :

0			

On ajoute le nombre 1 :

0	1		

On ajoute le nombre 3 :

0	1	2	

Il reste 1 case non utilisée.
Comme nous avons 3/4(4)=3 nombres dans la table, on doit étendre la table au double de sa capacité initiale. La capacité de la table est passée de 4 à 8. Soit n=8.

Pour ajouter le nombre 4, on doit d'abord copier les 3 éléments de l'ancienne table dans la nouvelle table comme suit :

0	1	2					

Maintenant, on peut ajouter le nombre 3 :

0	1	2	3				

On ajoute ensuite le nombre 4:

0	1	2	3	4			

On ajoute le nombre 5 :

0	1	2	3	4	5		

Il reste 2 cases non utilisées.
On constate qu'on a atteint les 3/4 d'éléments nécessaires dans la table, soit 3/4(8)=6. On doit à nouveau doubler la capacité du tableau, elle doit passer de 8 à 16.

On copie les éléments de la table précédente :

0	1	2	3	4	5										

Pour atteindre le 3/4 d'éléments dans la table, soit 12 éléments, on doit 6, 7, 8, 9, 10, 11. On obtient ainsi la table ci-après :

0	1	2	3	4	5	6	7	8	9	10	11				

Il reste 4 cases non utilisées.
Après l'ajout du nombre 11, on doit à nouveau étendre la table au double de sa capacité, soit 32.
On copie les éléments de l'ancienne table dans la nouvelle :

0	1	2	3	4	5	6	7	8	9	10	11																	

Il reste 2 cases non utilisées.
Pour atteindre le 3/4 d'éléments de la table, on doit au total avoir 24 éléments de 0 à 22.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23							

Il reste 8 cases non utilisées.

Conclusion
Si l'extension se déclenche lorsqu'on a rempli la table au 3/4 de sa capacité, le coût en espace n'est pas bon car il reste des cases inutilisées.

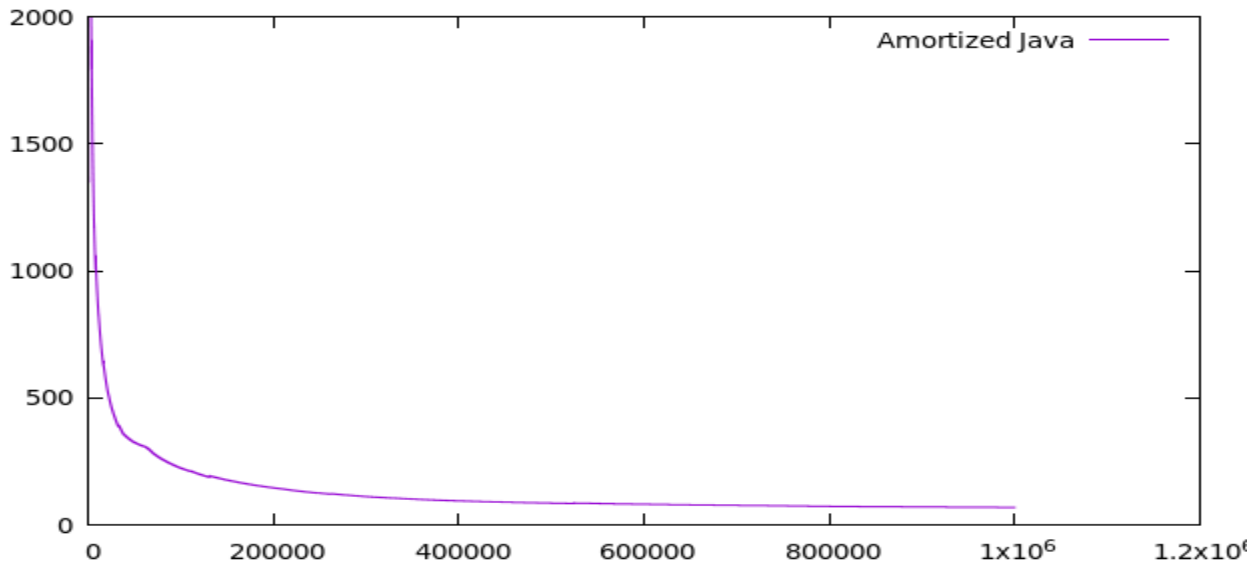
4) Pour la suite du TP, nous avons choisi de travailler en Java.

Pour que la fonction « do_we_enlarge_capacity » ne se déclenche que lorsque le tableau est plein, le nombre d'éléments dans le tableau doit être égale à la capacité du tableau.

```
private boolean do_we_need_to_enlarge_capacity() {
    //return size >= (capacity * 3)/4;
    return size==capacity;
}
```

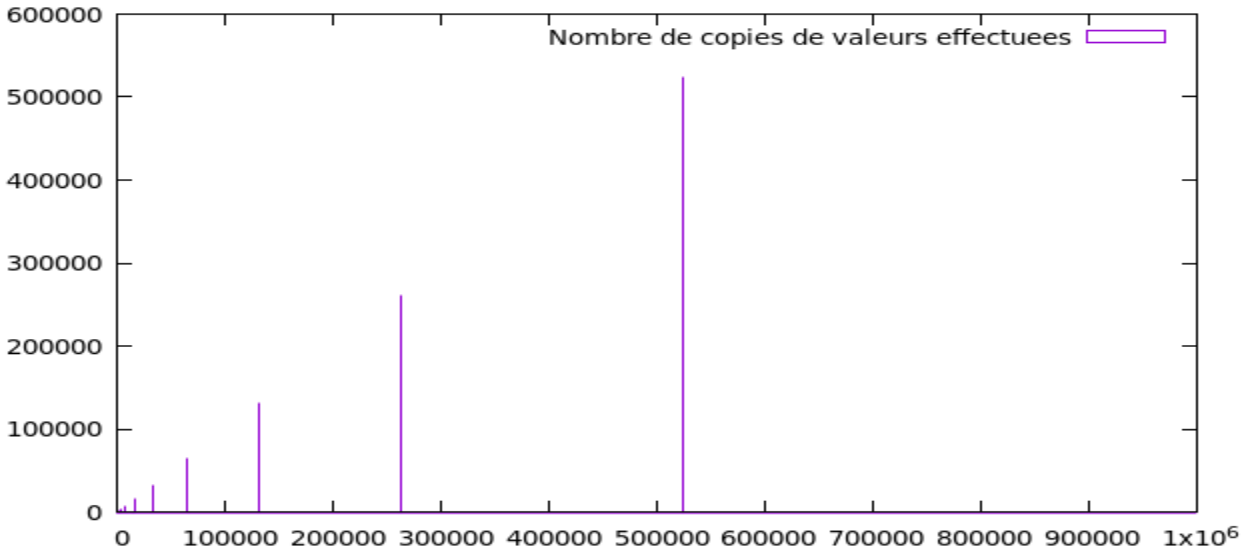
Expériences

Coût amorti



Le constat en est que le coût amorti n'augmente que pour les premières opérations. Après il reste constant jusqu'à l'exécution des $n=10^6$ opérations.

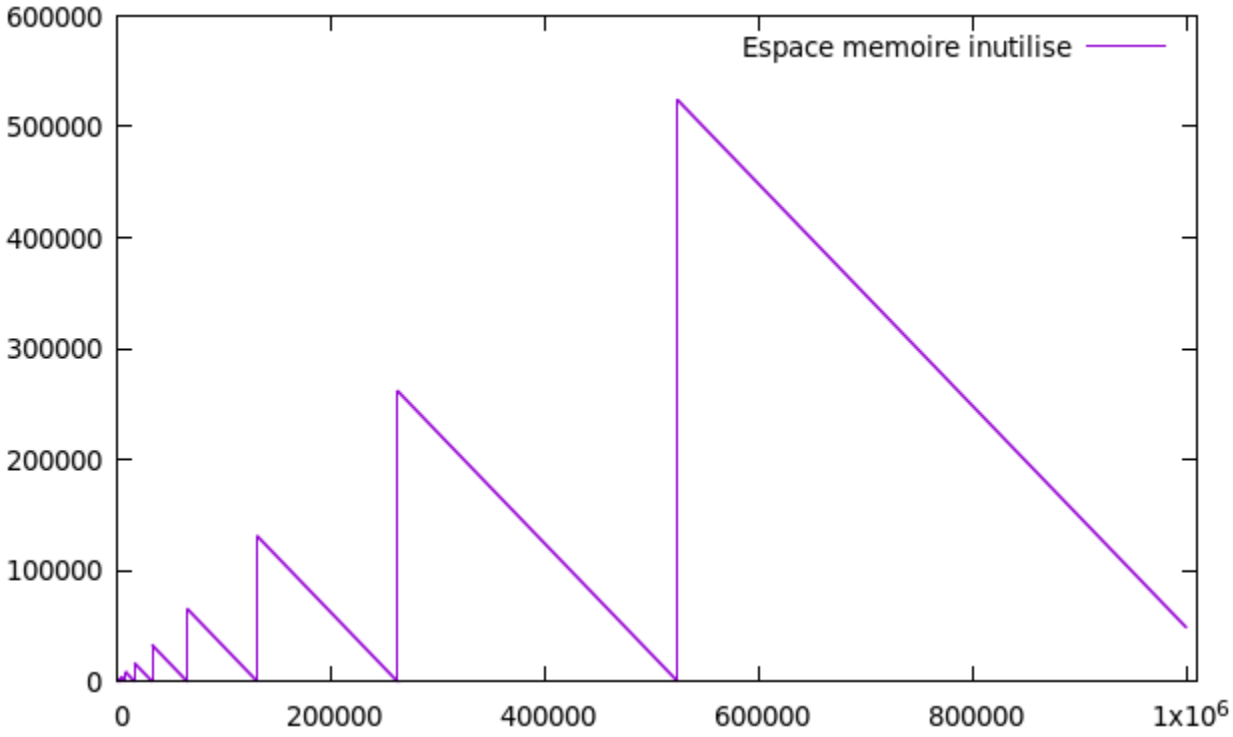
Nombre de copies



On constate qu'il y a :
- moins de pics d'augmentation du nombre de copies
- le pic maximal d'augmentation du nombre de copie est à près de 500000 copies contrairement au cas précédent où il était à près de 800000
- il y a copie que lorsque la table est pleine

En gros, il y a une réelle amélioration par rapport au cas précédent.

Mémoire inutilisée

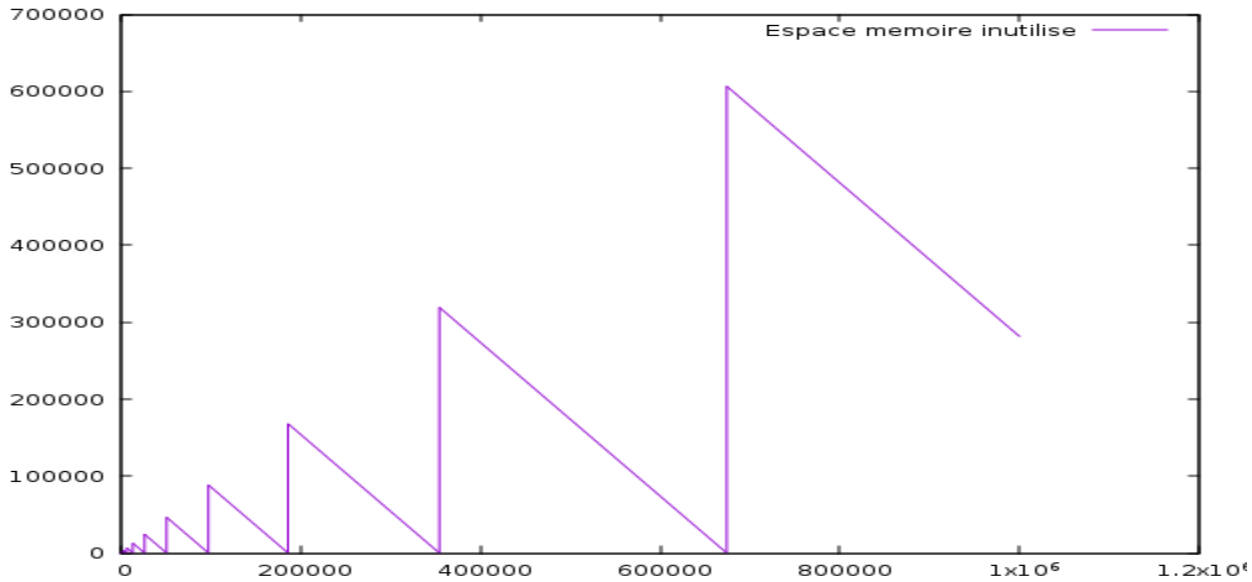


Nous constatons que :

- l’espace mémoire inutilisée n’augmente que lorsqu’il est nul c’est-à-dire quand on n’a plus d’espace mémoire dans la table
- l’augmentation de la mémoire inutilisée ne dépasse pas la capacité du tableau contrairement au cas précédent
- il reste environ 50 000 cas non utilisés

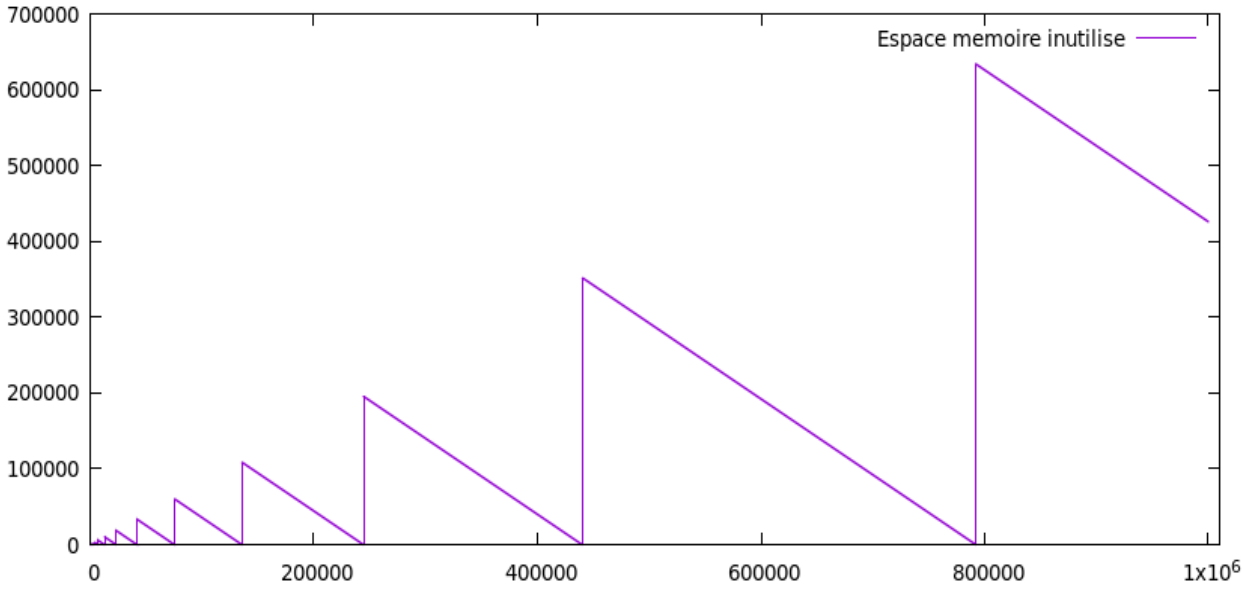
5)
Variation du facteur multiplicatif α

Pour α =1.9



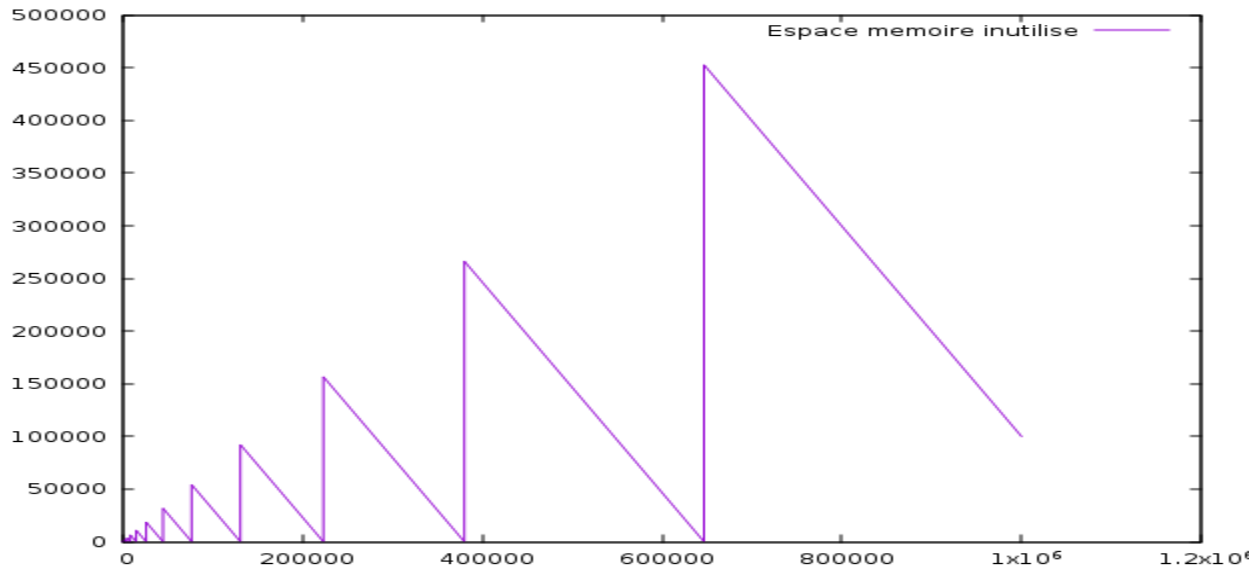
il reste environ 300000 cases non utilisés.

Pour α =1.8



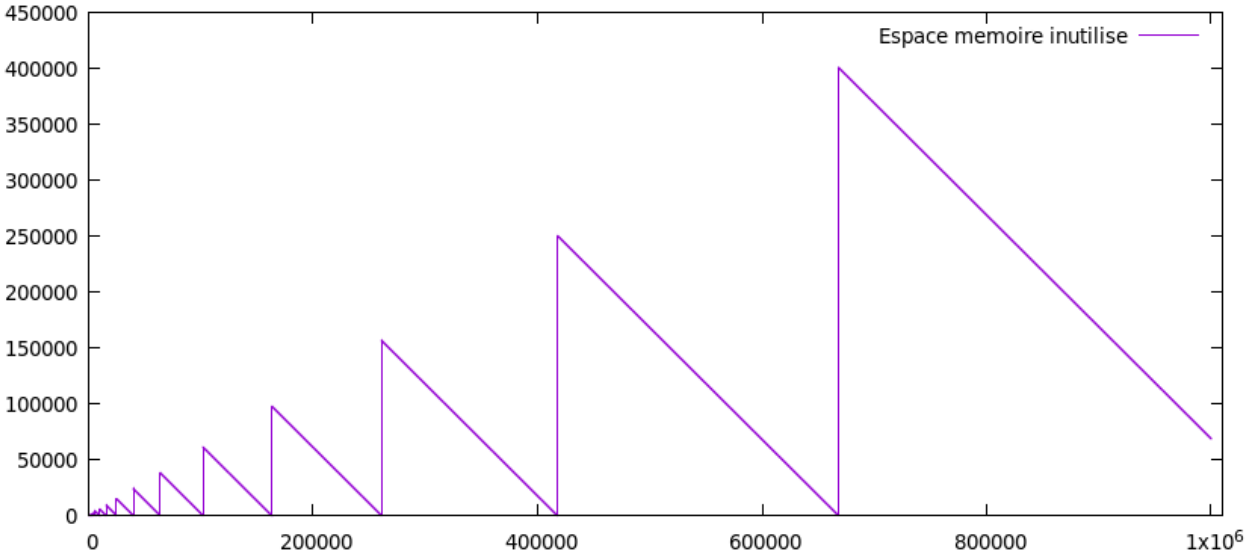
Il reste environ 400000 cases non utilisés.

Pour α =1.7



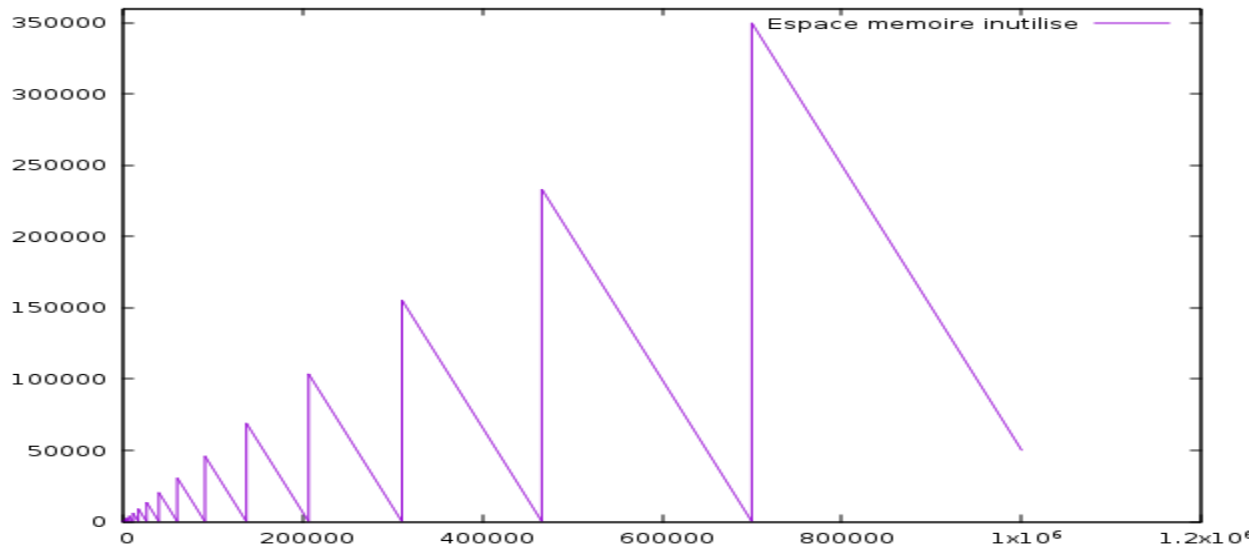
Il reste environ 100000 cases non utilisés.

Pour α =1.6



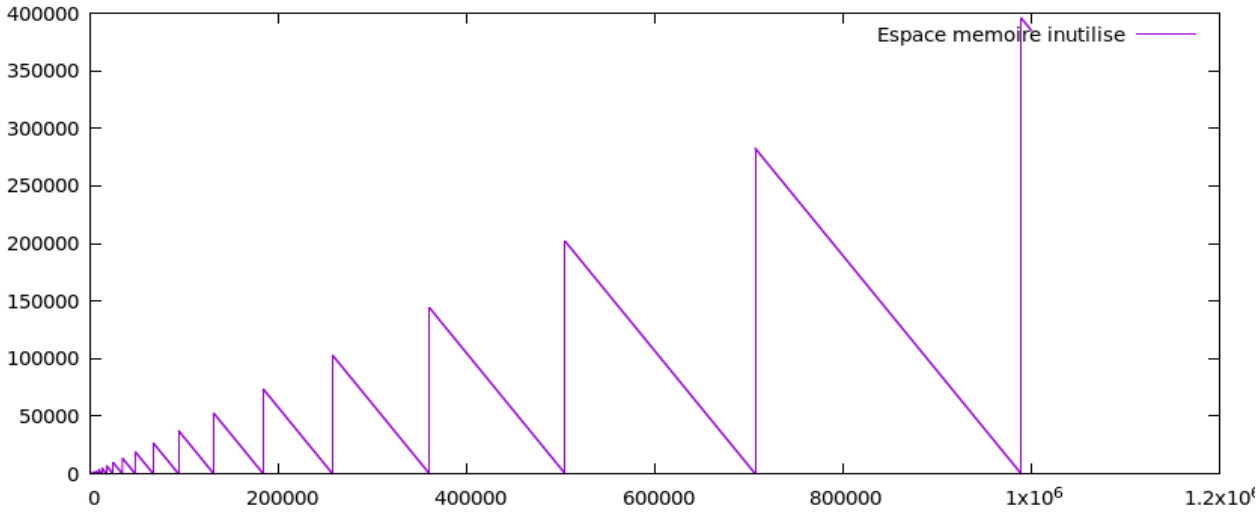
Il reste environ 400000 cases non utilisés.

Pour α =1.5



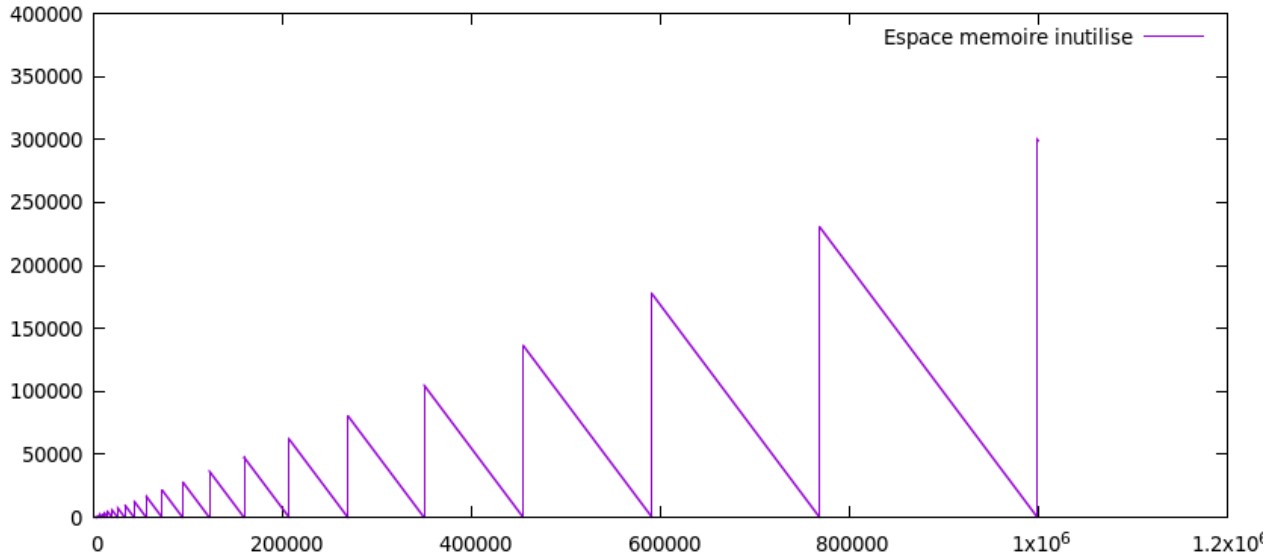
Il reste environ 50000 cases non utilisées.

Pour α =1.4



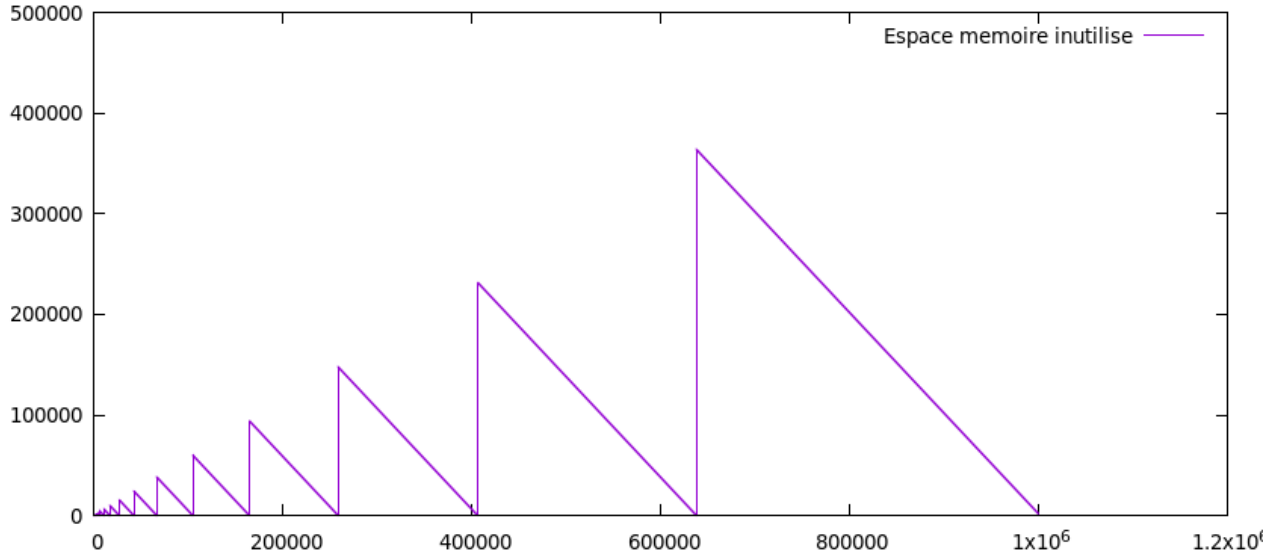
Il reste environ 350000 cases non utilisés.

Pour α =1.3



Il reste environ 300000 cases non utilisés.

Pour α =1.57



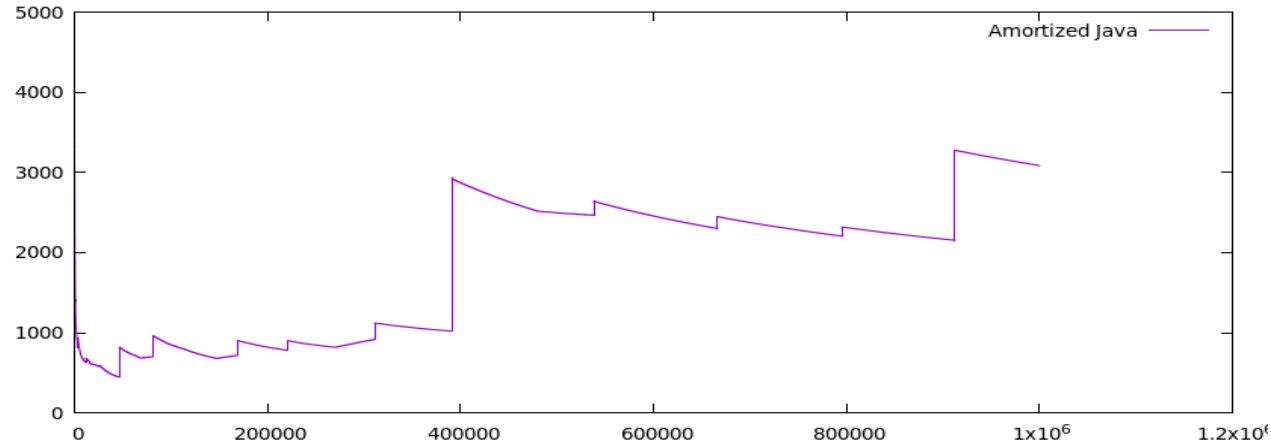
Nous constatons que toutes les cases sont utilisés. Il ne reste pas d'espace mémoire non utilisée.

Règle décrivant le rapport entre le coût en temps et le coût en espace

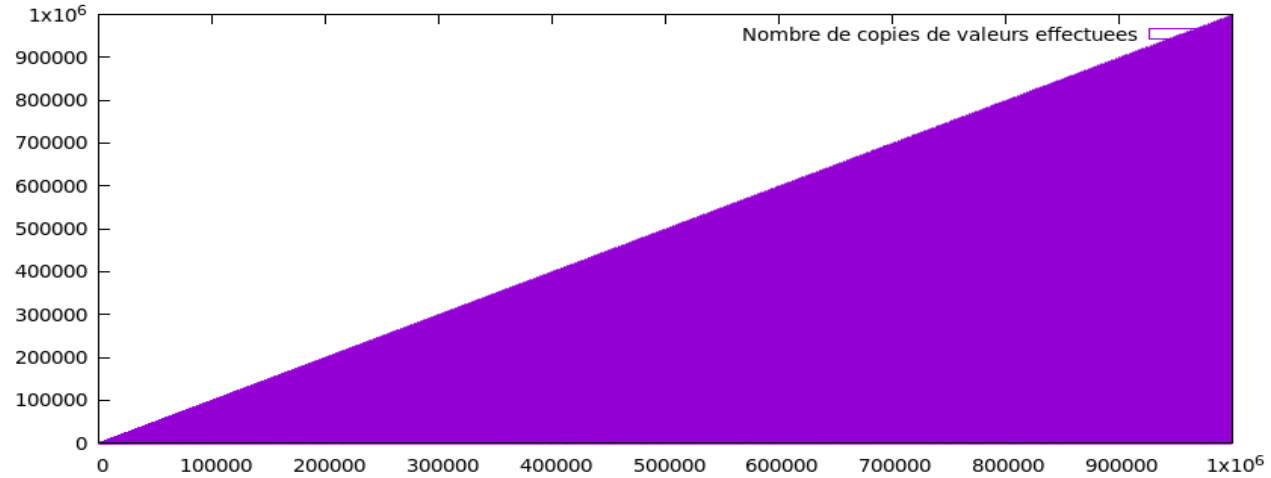
Quand l'espace mémoire non utilisée est élevé, le coût en espace est élevé et le coût en temps est élevé.
D'autre part, quand l'espace mémoire non utilisée est faible, le coût en espace est faible et le coût en temps est faible.

6)
Quand on fait varier la capacité n vers une capacité n + √n, l'expérience nous donne :

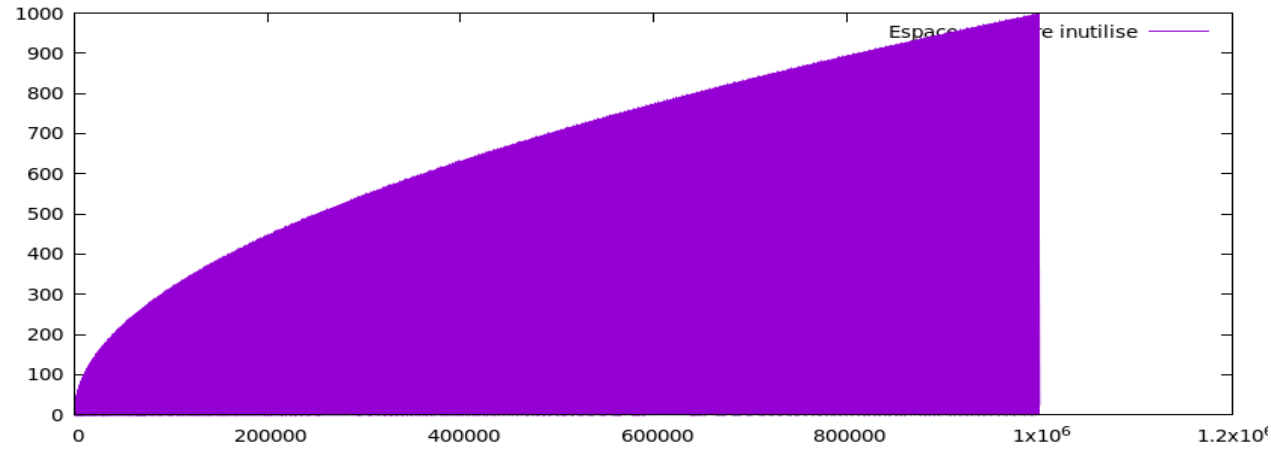
Coût amorti :



Nombre de copies



Mémoire inutilisée



Conclusion
En faisant cette expérience, on constate que le coût des opérations augmente davantage. Le coût amorti augmente au début et reste élevé au début et lors des opérations ultérieures. Le gaspillage de mémoire est moins important. Le nombre de copies est important au début et relativement faible ultérieurement.

Synthèse

A travers ce TP, nous avons fait une analyse amortie dans une table dynamique. Il nous a permis d'identifier les bonnes et mauvaises techniques quand on fait une analyse amortie dans une table dynamique.
Le choix du langage est essentiel quand on écrit un programme, en effet dans la catégorie des langages, les langages compilés (C/C++) sont plus rapides que les langages interprétés (Java/Python). En revanche, d'un ordinateur à un autre, on doit (en général) recompiler un programme écrit en langages compilés pour l'exécuter, ce qui n'est pas le cas des langages interprétés car les exécuter directement.
Quand on exécute plusieurs fois un même programme, le coût varie d'une exécution à une autre, cela est dû au fait qu'il y a d'autres programmes qui s'exécutent en même temps.
Quand on fait une analyse amorti, il est judicieux de commencer par identifier la (ou les) partie(s) du programme qui prend le plus de temps à s'exécuter.
Ensuite, il est nécessaire de choisir le bon facteur multiplicatif et le bon moment pour étendre la table.

- ➔ Quand on étend la table avant qu'elle ne soit pleine, on enregistre un gaspillage important de la mémoires, disons qu'on se retrouve avec beaucoup de cases mémoires non utilisés.
- ➔ En revanche, quand on étend la table que lorsqu'elle est pleine, on obtient un meilleur coût en espace.

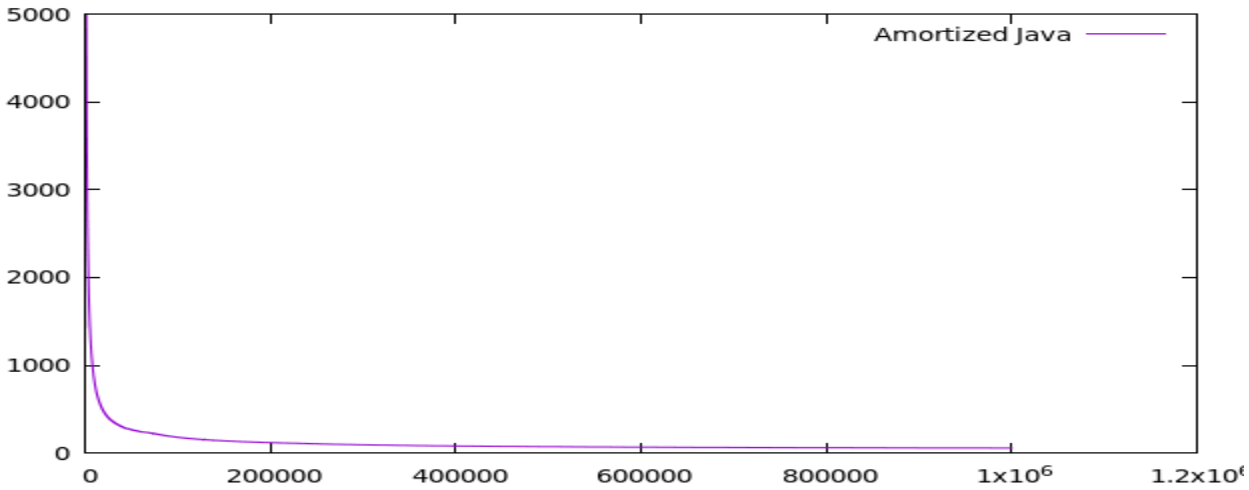
Cette extension, qui implique une réallocation mémoire, nous emmène à choisir un facteur multiplicatif. En faisant varier le facteur multiplicatif de 1.3 à 1.9, on obtient un meilleur coût en espace quand le facteur multiplicatif est égale à 1.57. Au dessus du facteur 2, on constate qu'on fait certes moins de réallocation mémoire mais coût en espace n'est pas meilleur.

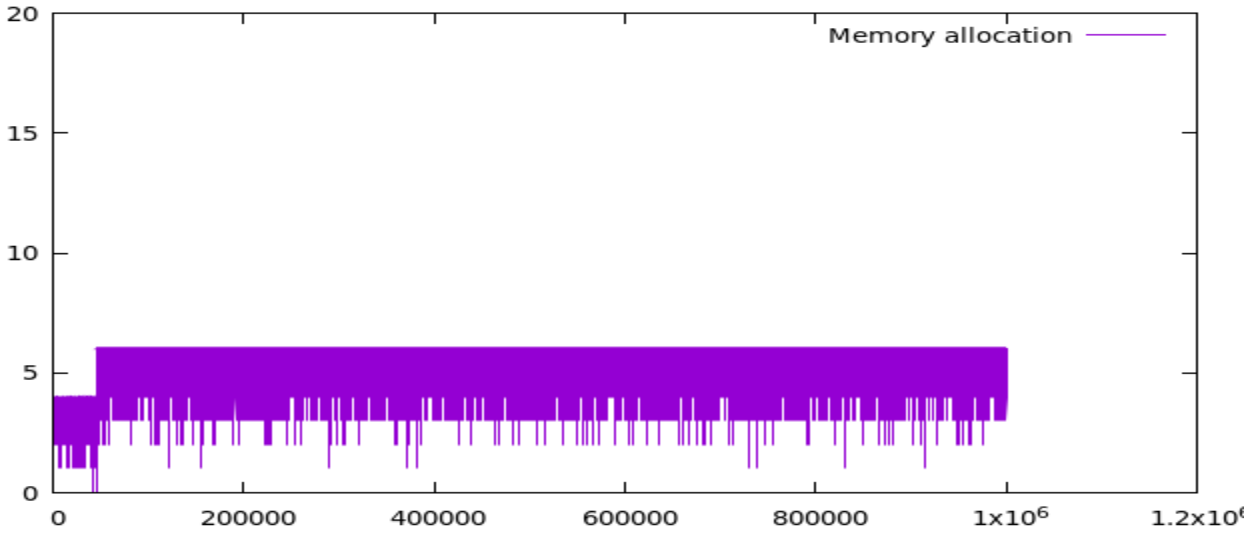
Donc, pour faire une bonne analyse amorti dans une table dynamique et obtenir les bons résultats, il faut étendre la table quand elle est pleine et multiplier cette table par un facteur qui minimise le coût en espace, de ce fait c'est le facteur 1.57 qui est meilleur.

TP2: INSERTIONS ET SUPPRESSIONS DANS UNE TABLE DYNAMIQUE

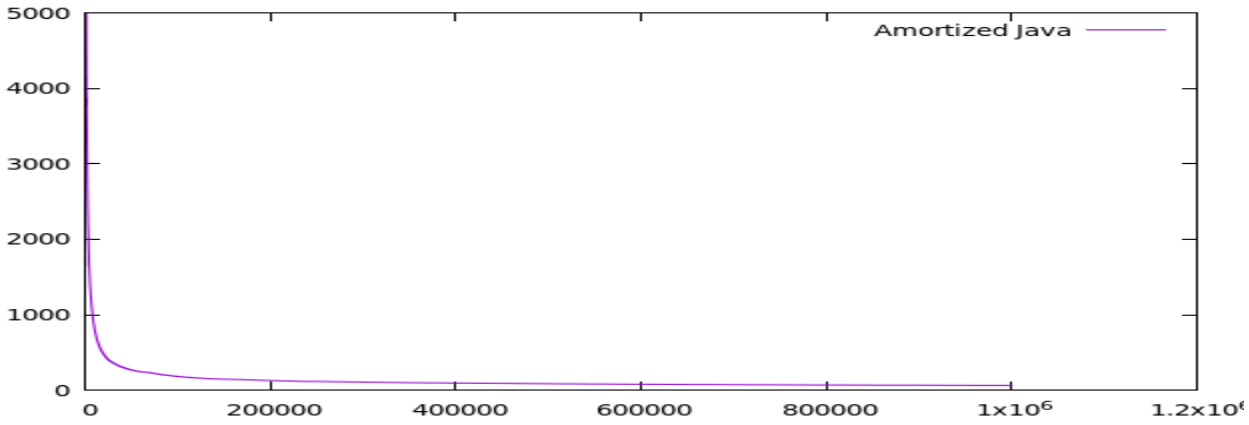
4)Résultats exécutions pour :

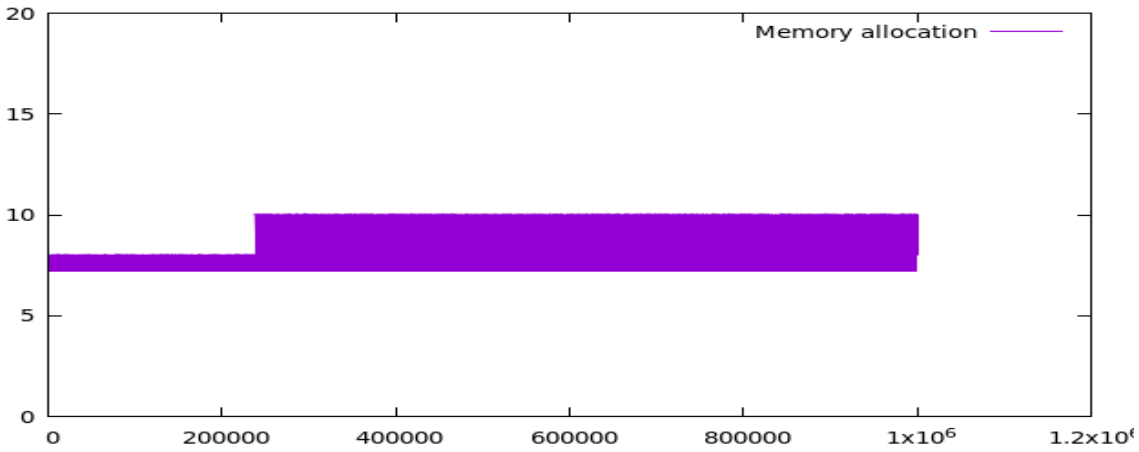
p=0.1



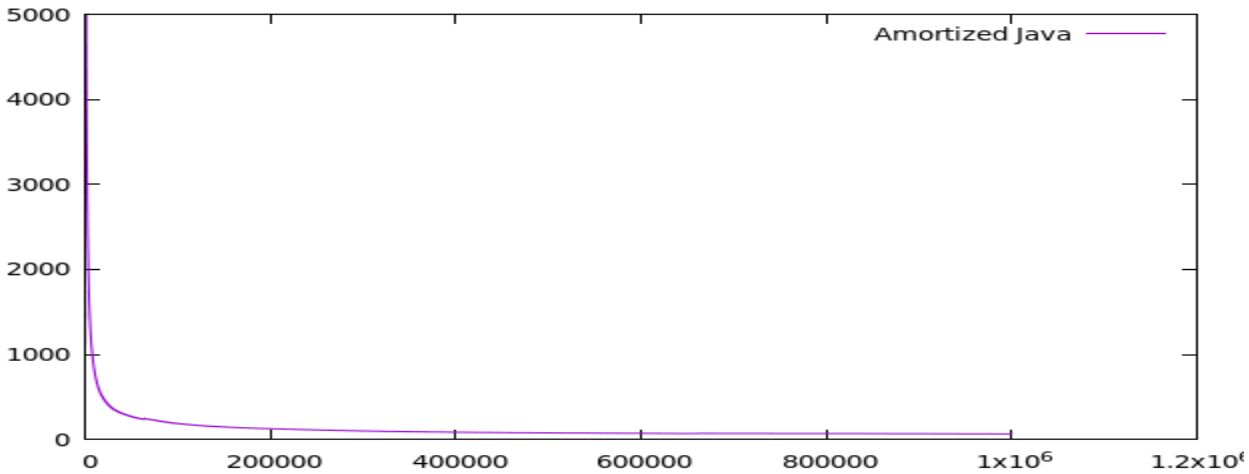


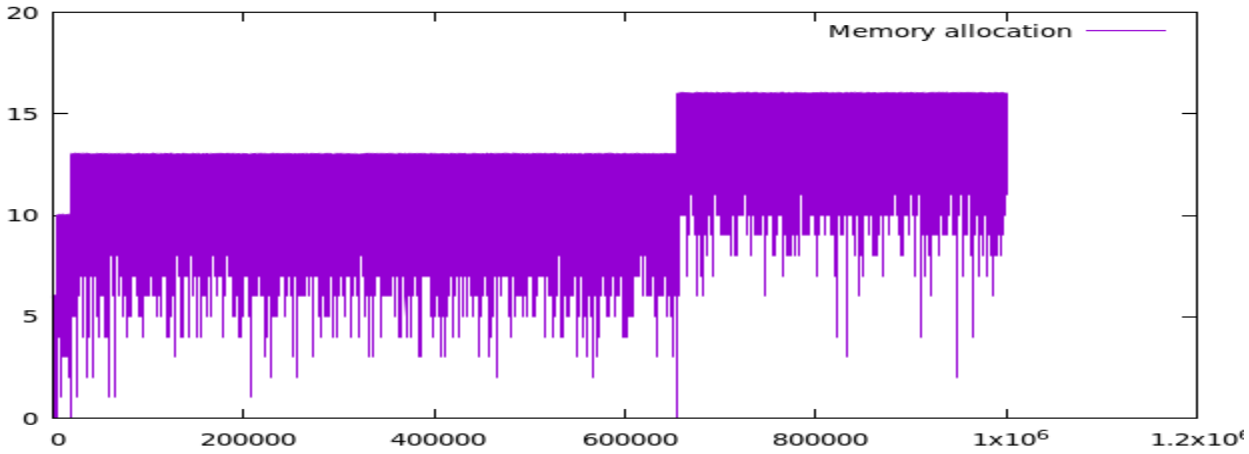
p=0.2


Amortized Java
5000
4000
3000
2000
1000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

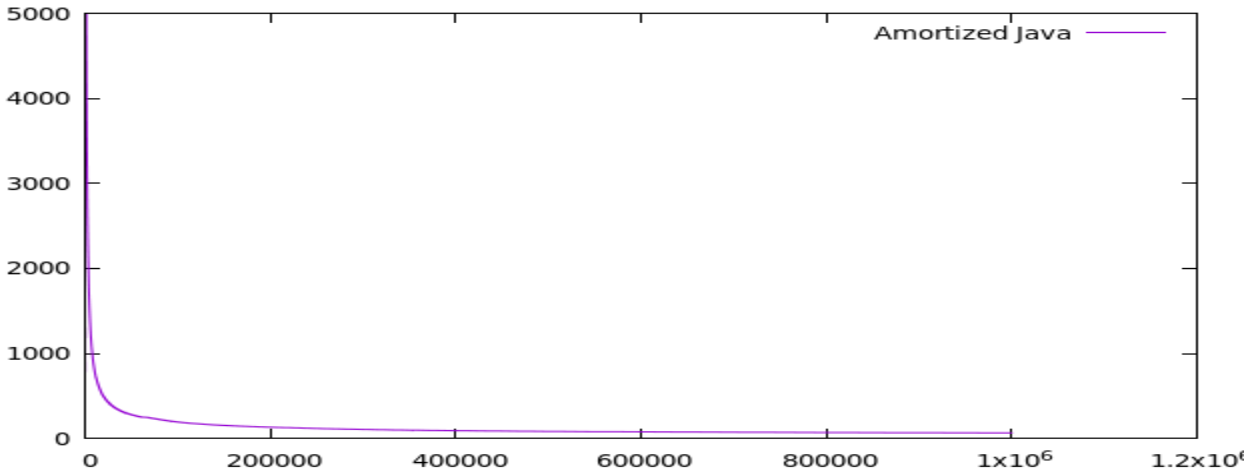

Memory allocation
20
15
10
5
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

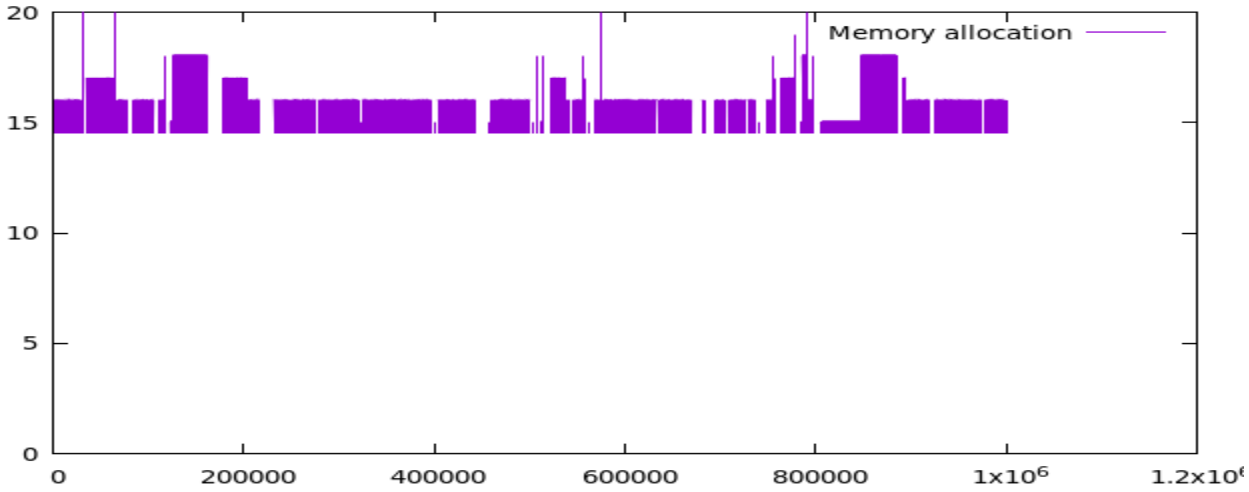
p=0.3


Amortized Java
5000
4000
3000
2000
1000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

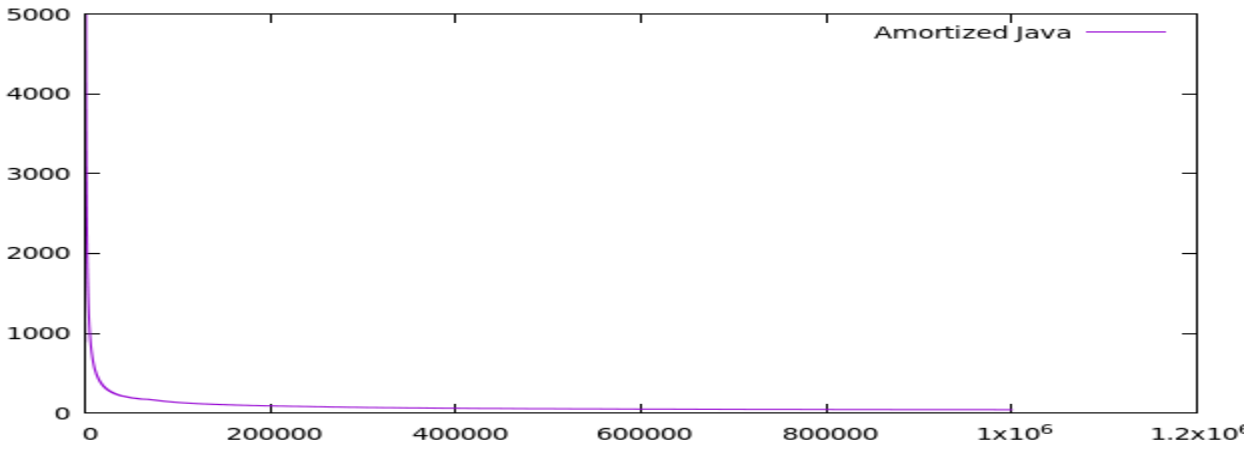

Memory allocation
20
15
10
5
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

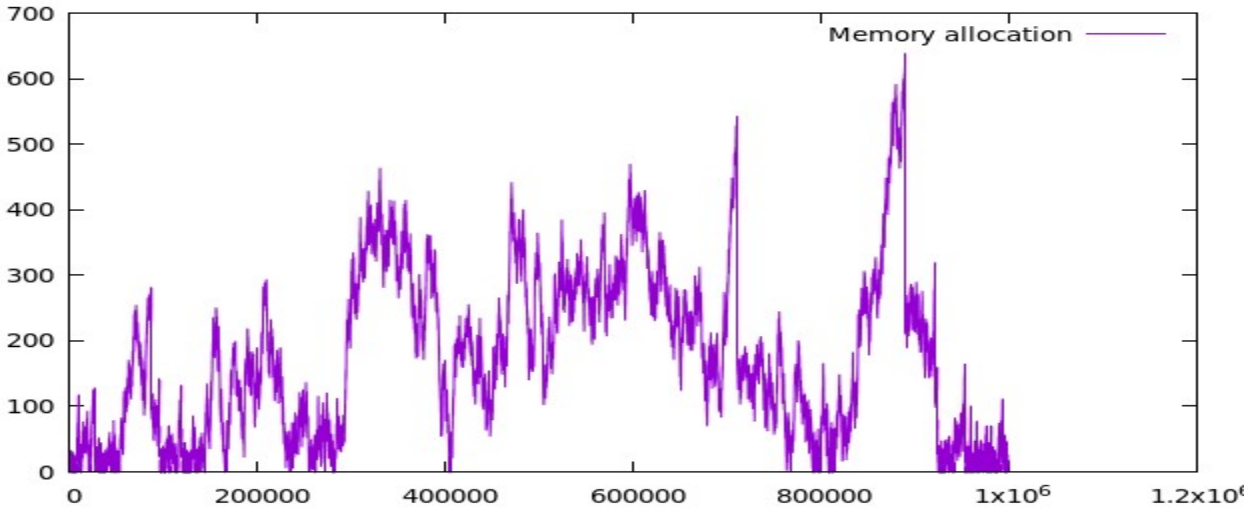
p=0.4


Amortized Java
5000
4000
3000
2000
1000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

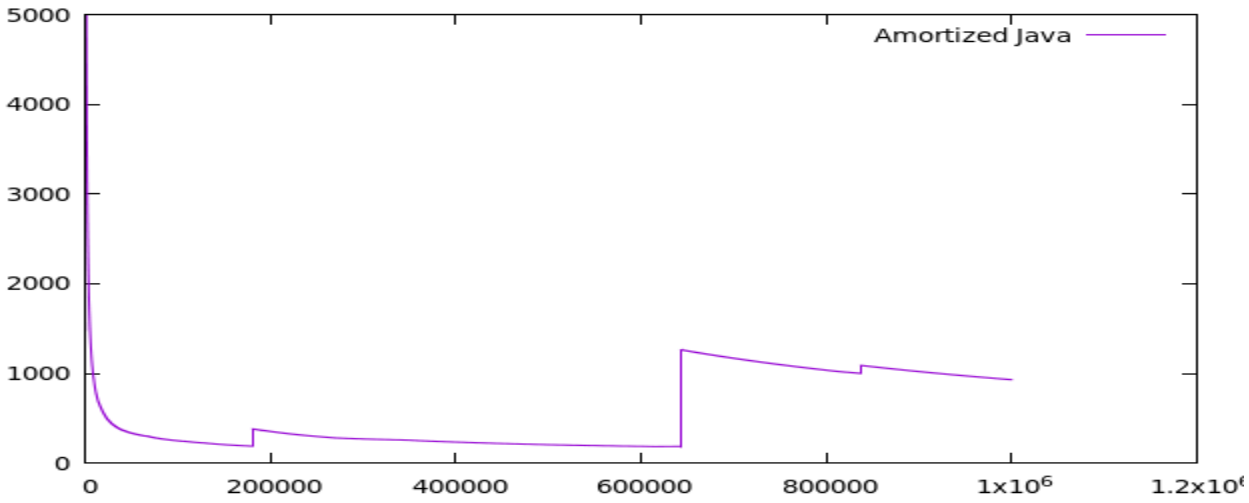

Memory allocation
20
15
10
5
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

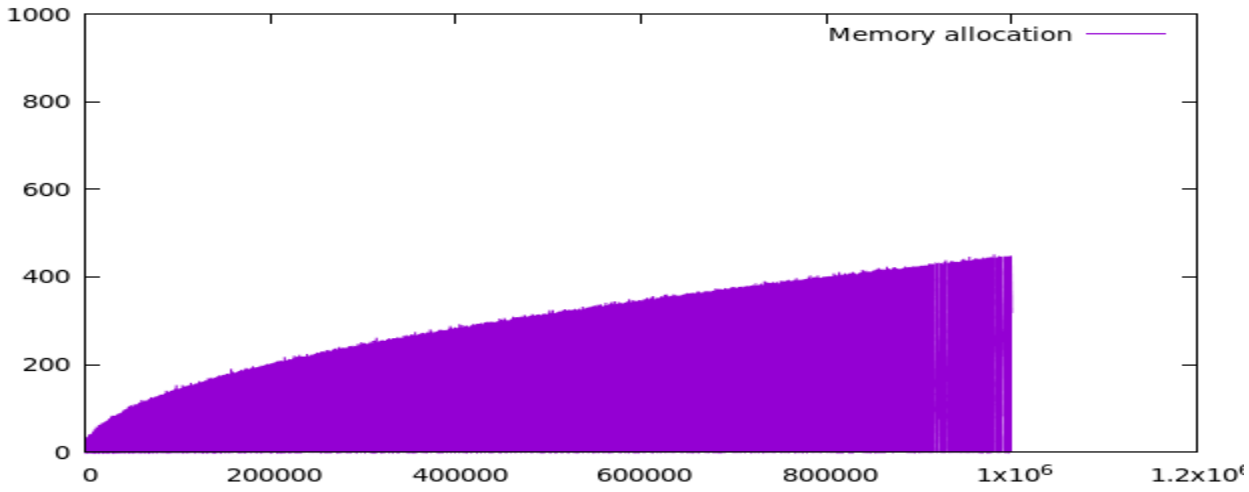
p=0.5


Amortized Java
5000
4000
3000
2000
1000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

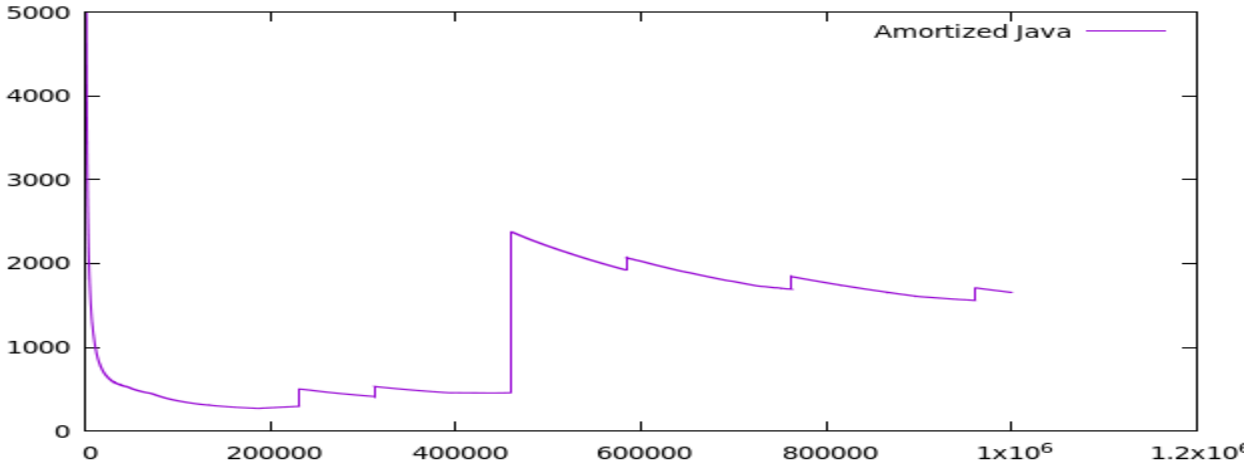

Memory allocation
700
600
500
400
300
200
100
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

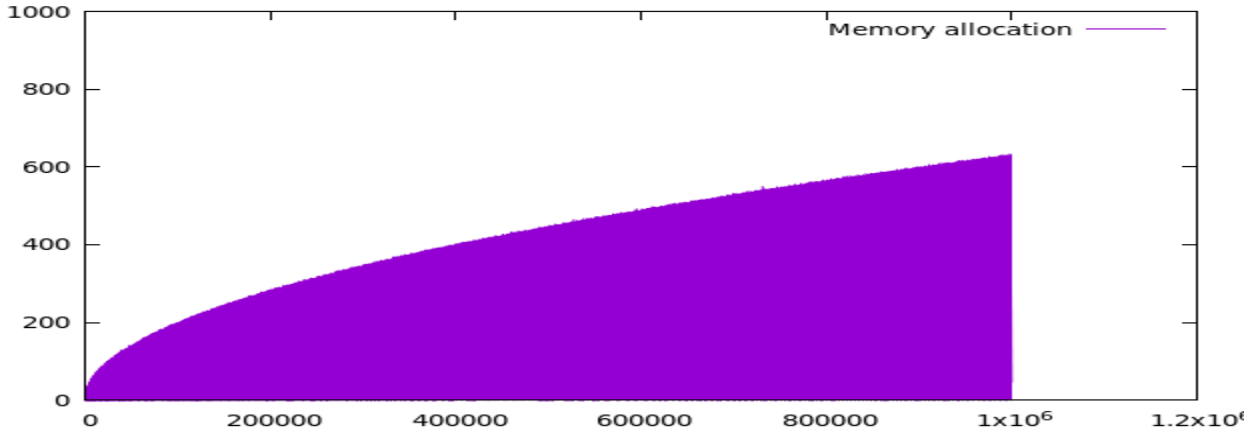
p=0.6


Amortized Java
5000
4000
3000
2000
1000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

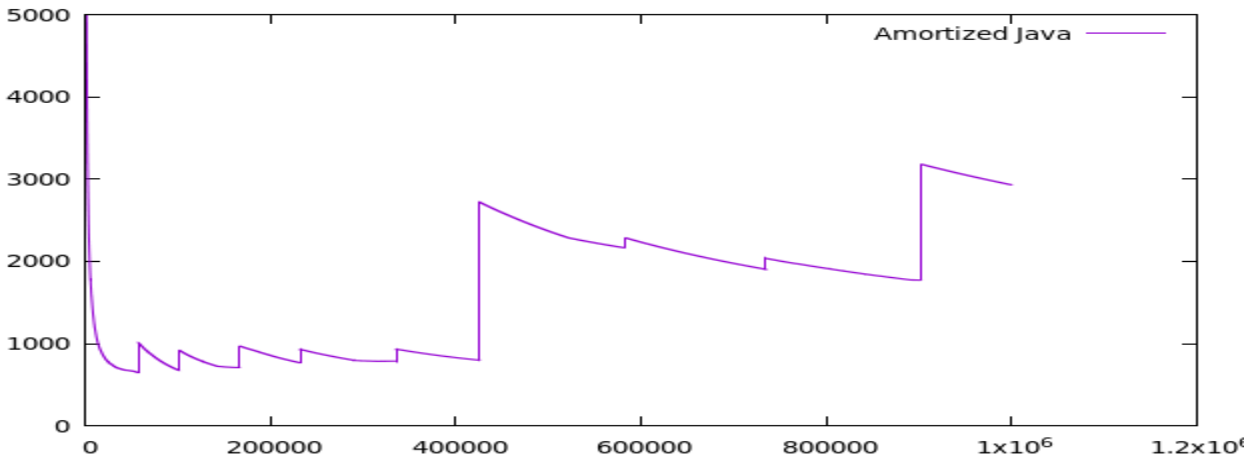

Memory allocation
1000
800
600
400
200
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

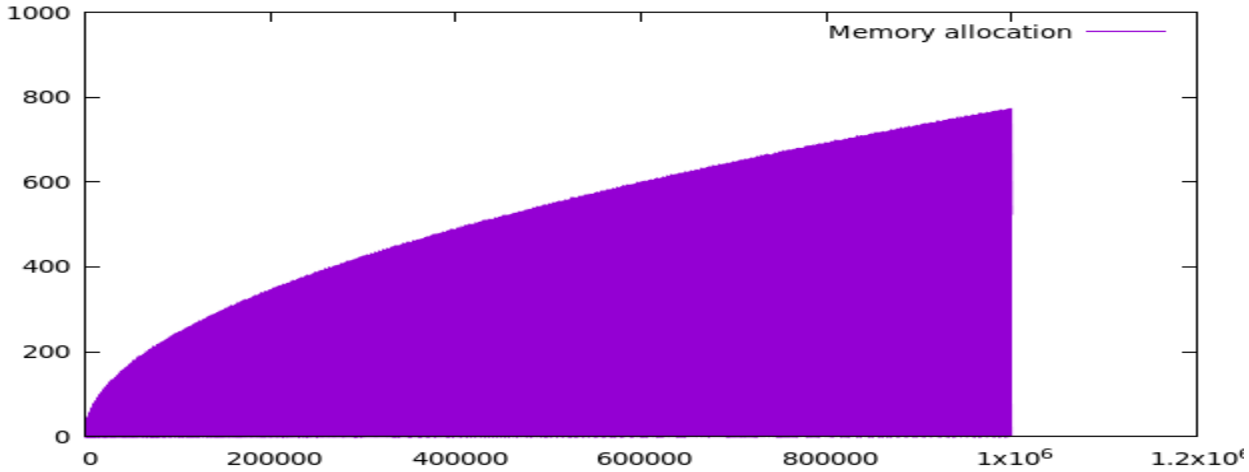
p=0.7


Amortized Java
5000
4000
3000
2000
1000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

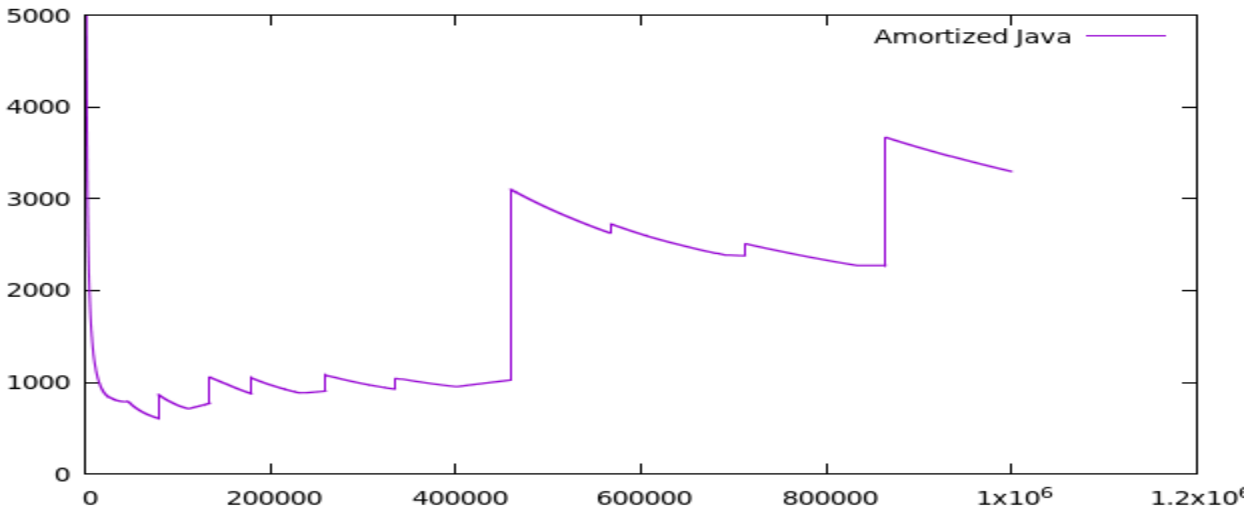

Memory allocation
1000
800
600
400
200
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

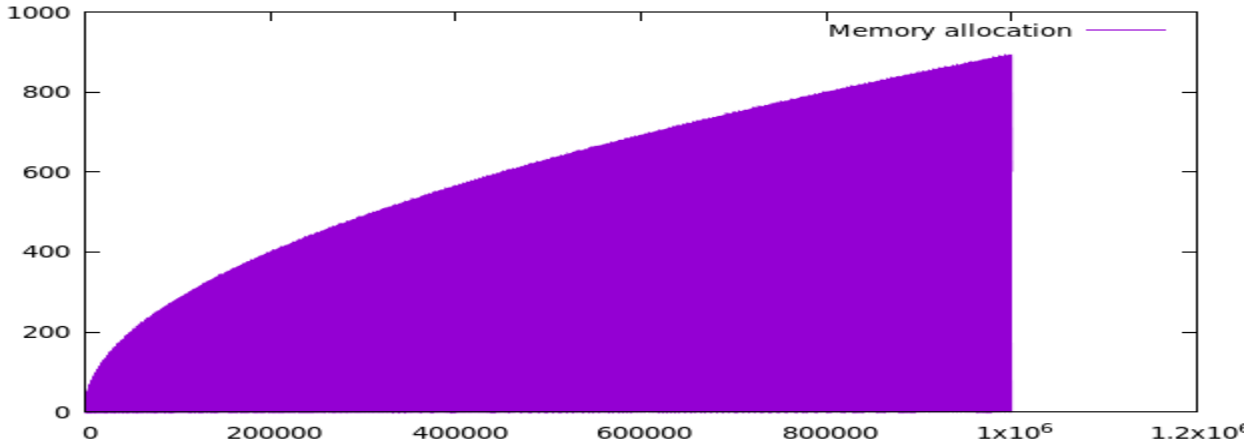
p=0.8


Amortized Java
5000
4000
3000
2000
1000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6


Memory allocation
1000
800
600
400
200
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

p=0.9


Amortized Java
5000
4000
3000
2000
1000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6



Relation entre p, le coût amorti et la mémoire inutilisée :
Après avoir tester différentes valeurs de p (de 0.1 à 0.9),
Nous constatons :

* Quand p<=0.5 , le gaspillage de mémoire et le coût amorti sont faibles.

* En revanche, quand p>0.5, le gaspillage de mémoire et le coût amorti sont importants.

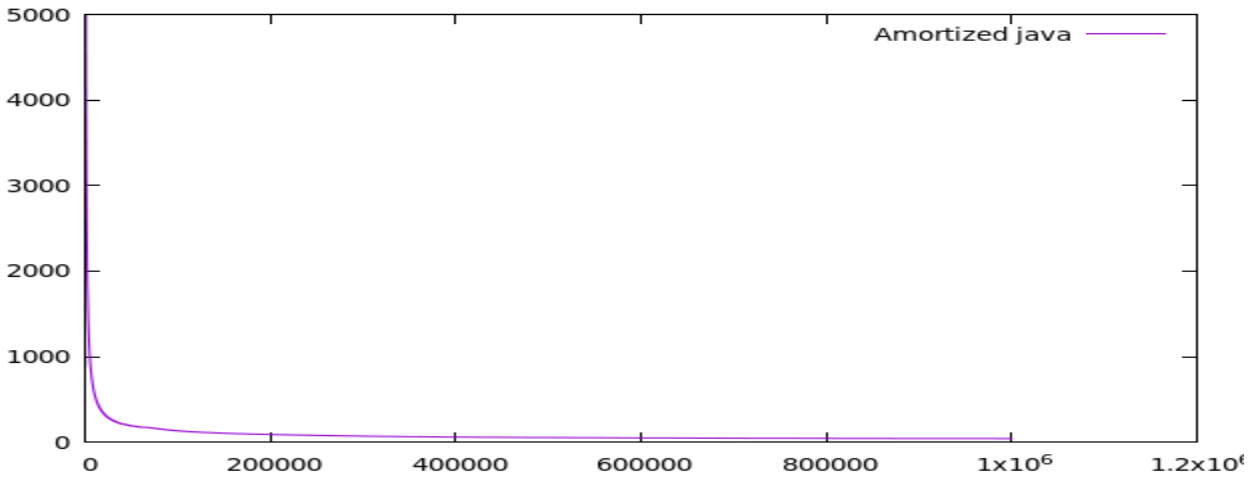
5)
Moment de contraction de la tableau
Il faut contracter la table quand le nombre d'éléments dans la table est inférieur à la moitié de la capacité du tableau.

```
private boolean do_we_need_to_reduce_capacity(){
    //return size <= capacity/4 && size >4;
    return size < capadity/2;
}
```

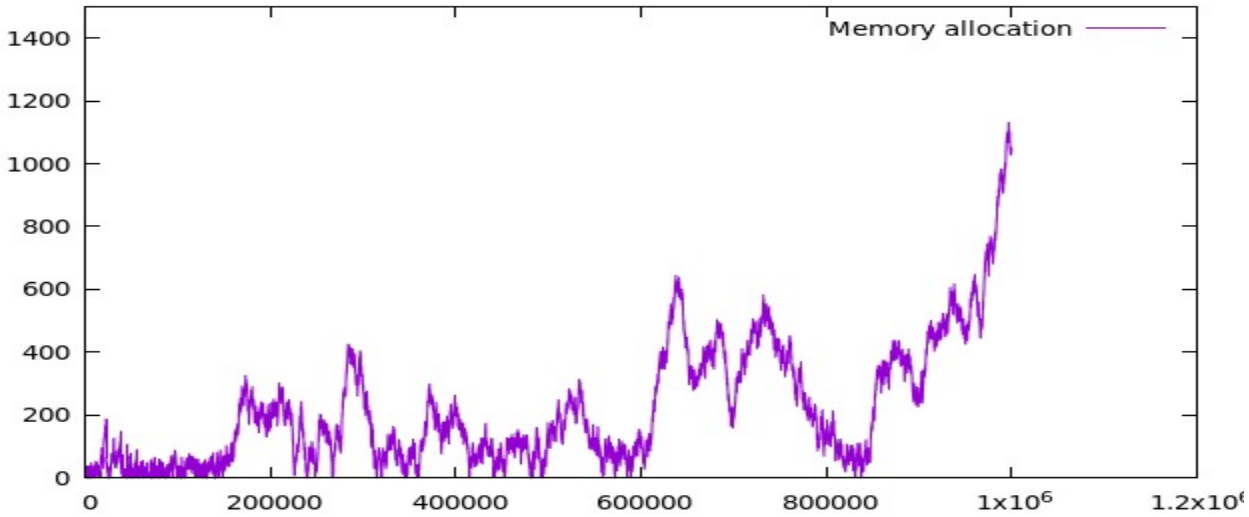
Efficacité de la nouvelle stratégie

Pour p=0.5

Coût amorti



Mémoire inutilisée

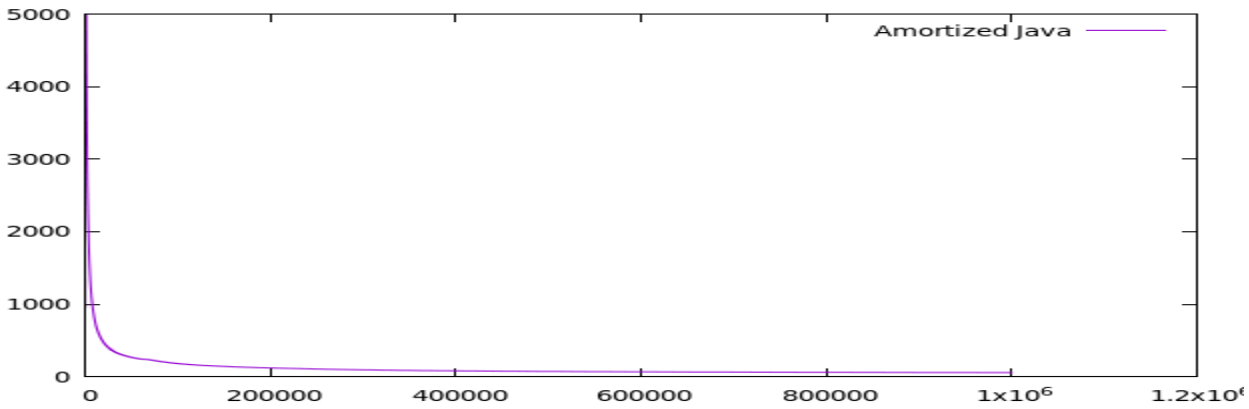


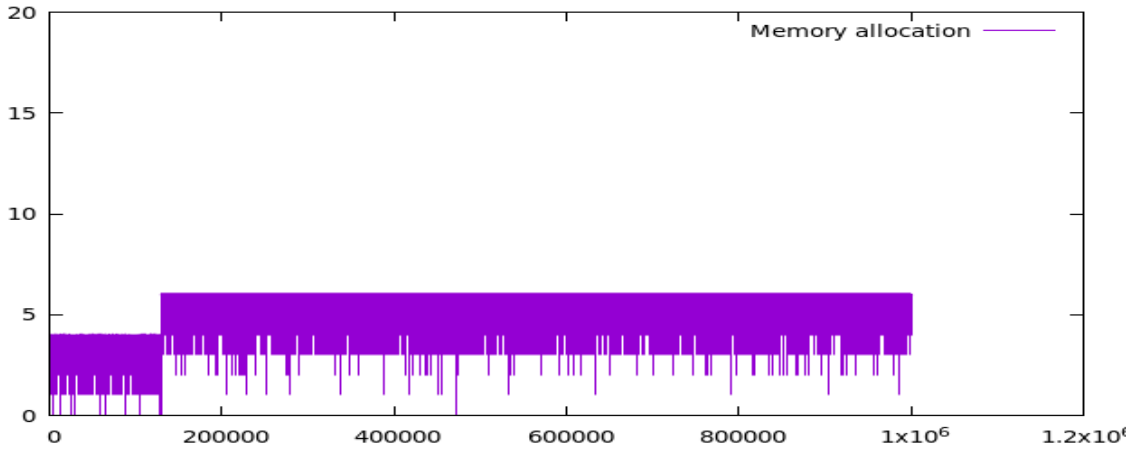
Avec cette nouvelle stratégie, le coût amorti et le gaspillage de mémoire ne se sont pas du tout améliorés, bien au contraire. En effet, le coût amorti est davantage plus grand en début. Quant au gaspillage de mémoire, il est encore plus important que dans la stratégie précédente.

Du coup, pour p=0.5, cette stratégie n'est pas du tout efficace.

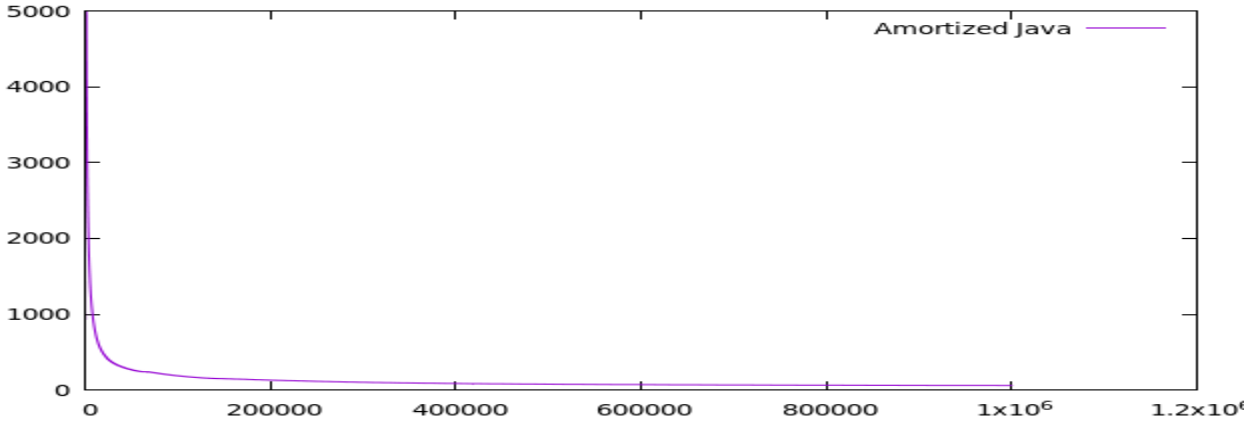
6)

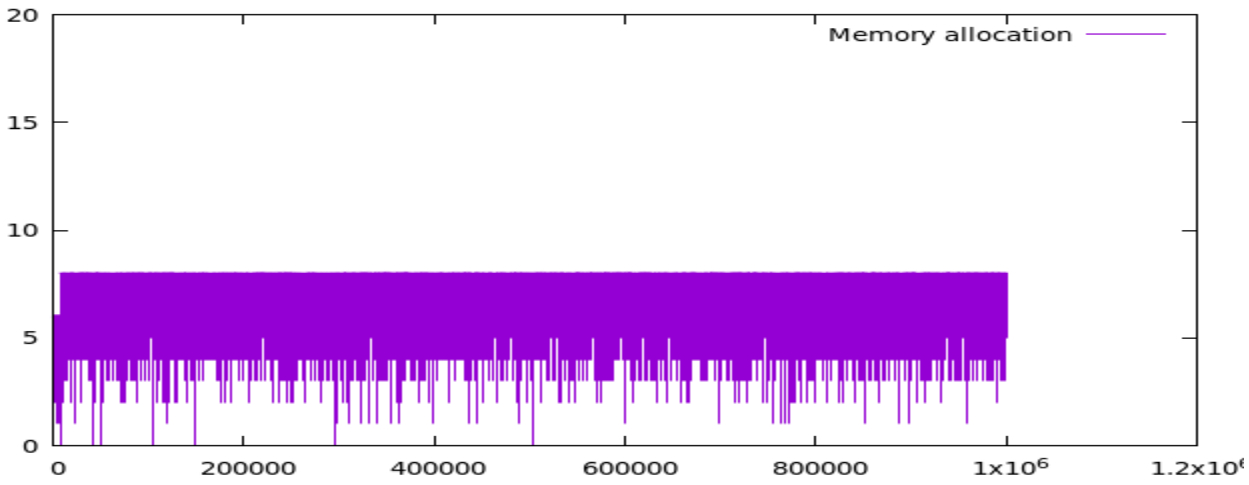
p=0.1


Amortized Java
5000
4000
3000
2000
1000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

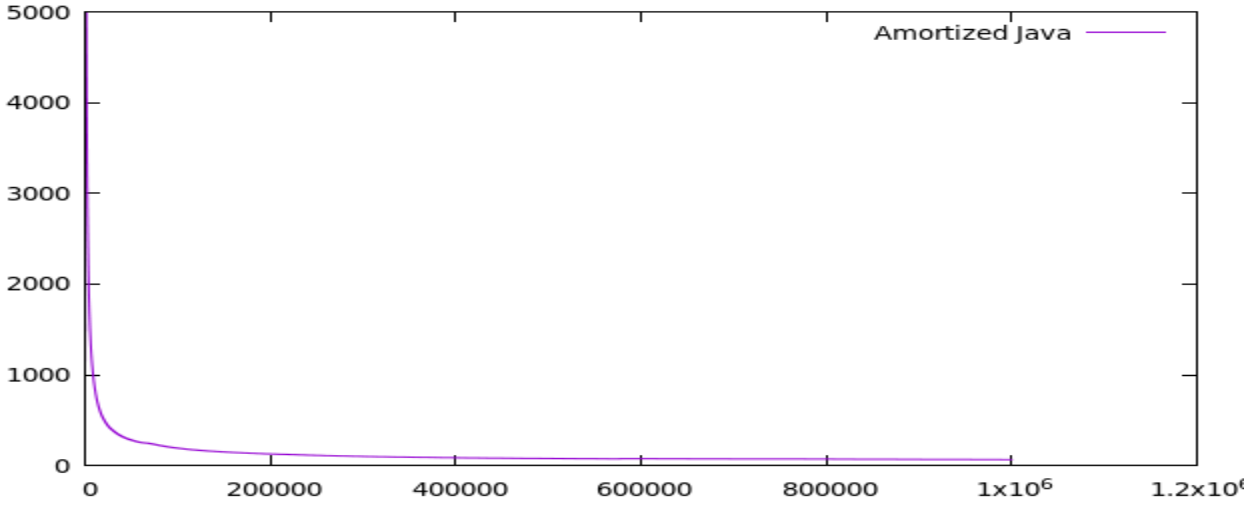

Memory allocation
20
15
10
5
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

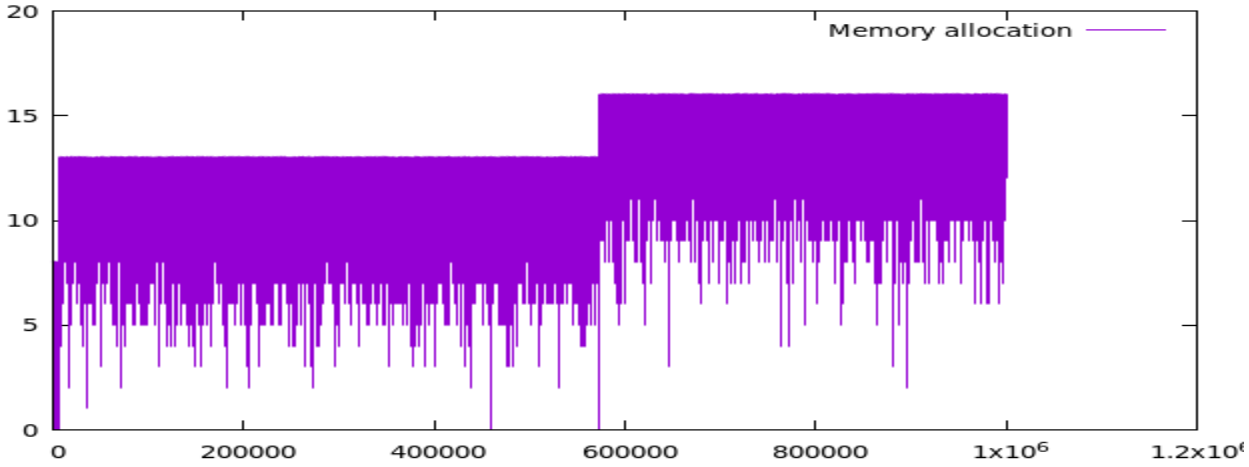
p=0.2


Amortized Java
5000
4000
3000
2000
1000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

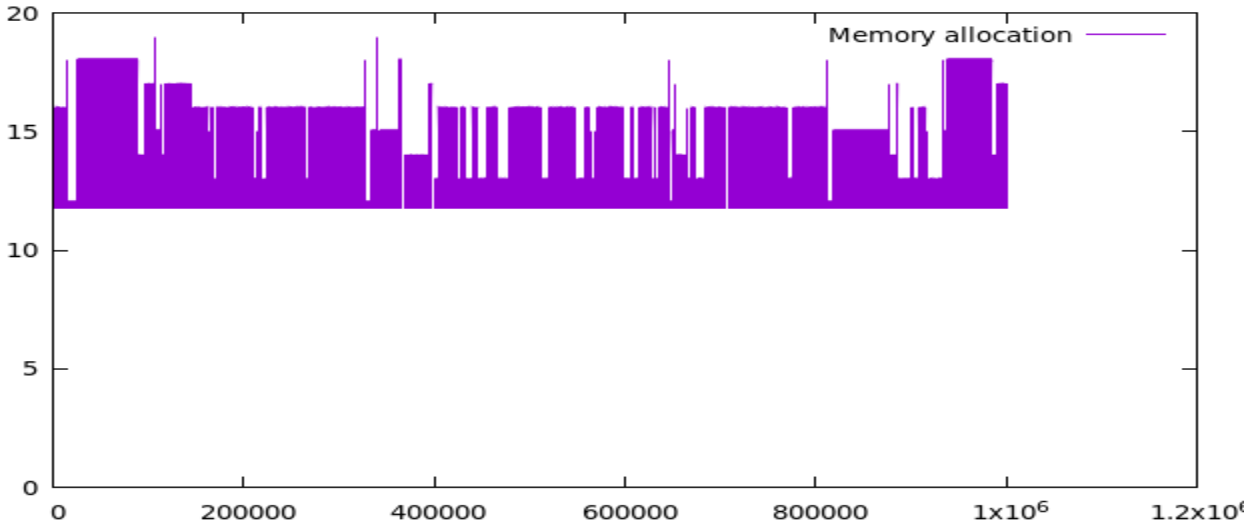

Memory allocation
20
15
10
5
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

p=0.3

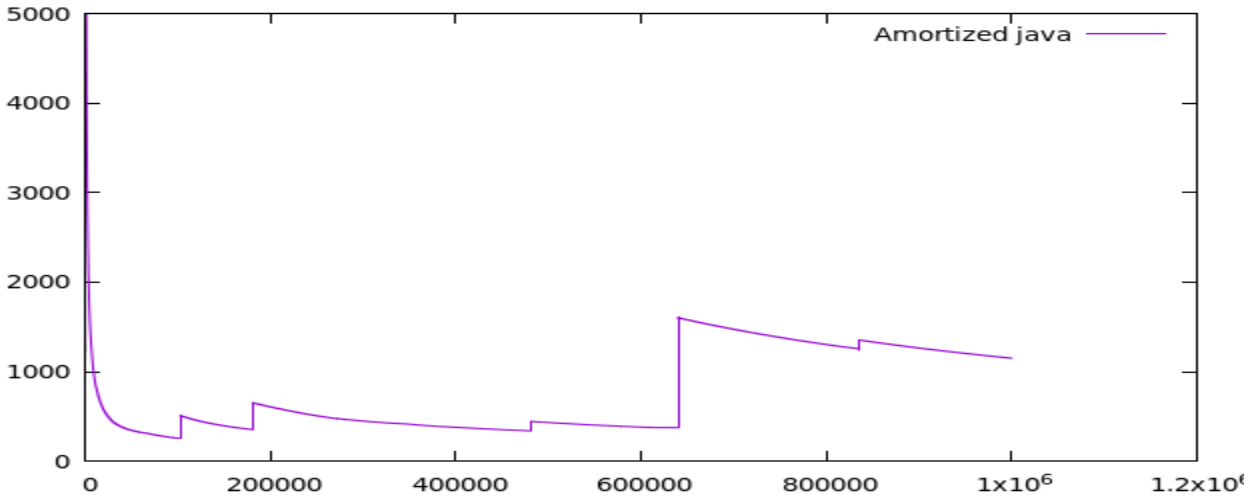

Amortized Java
5000
4000
3000
2000
1000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

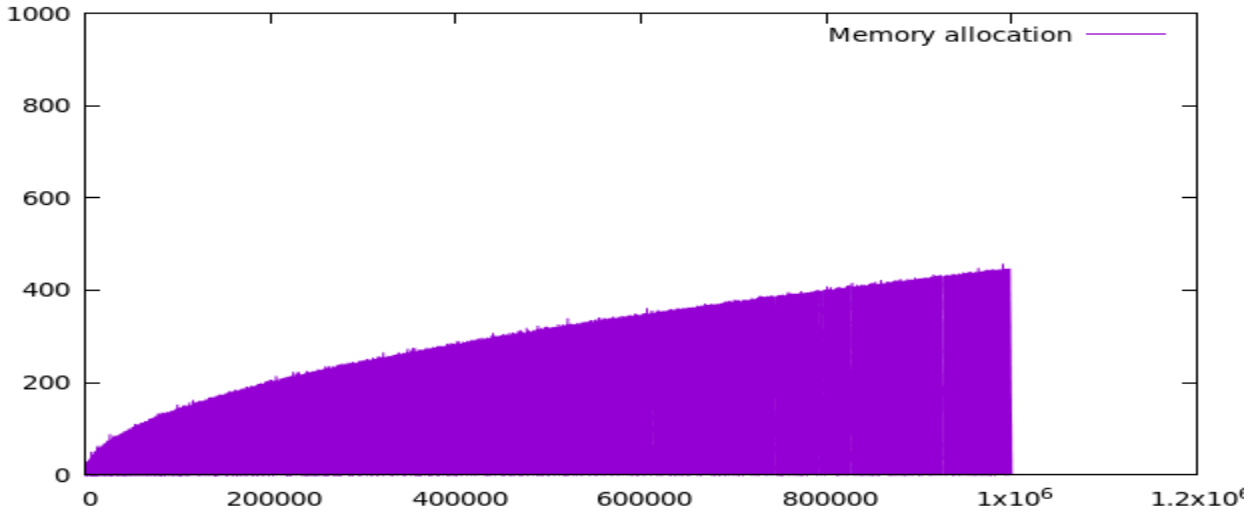

Memory allocation
20
15
10
5
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

p=0.4

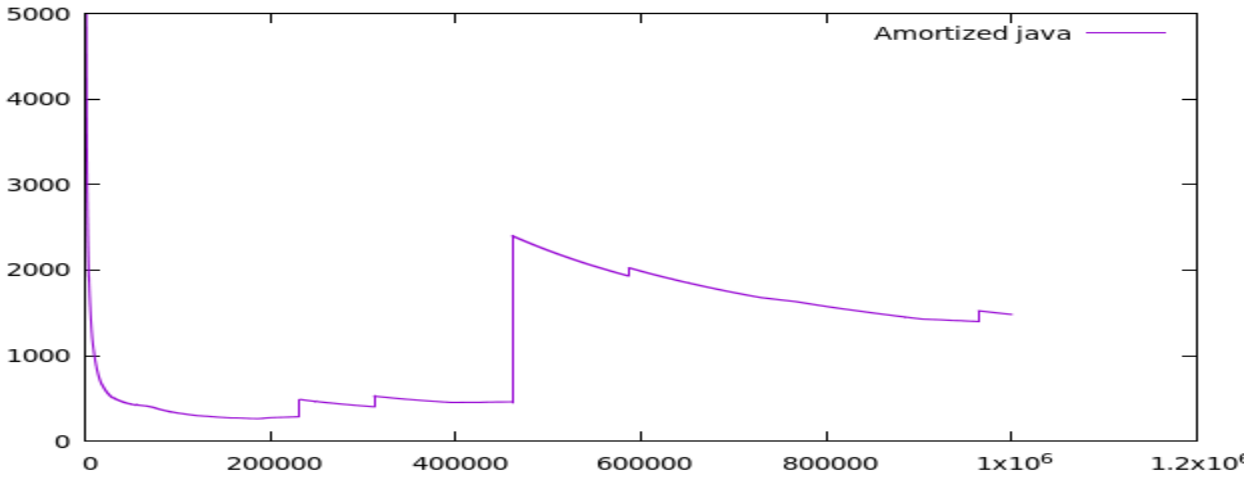

Memory allocation
20
15
10
5
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

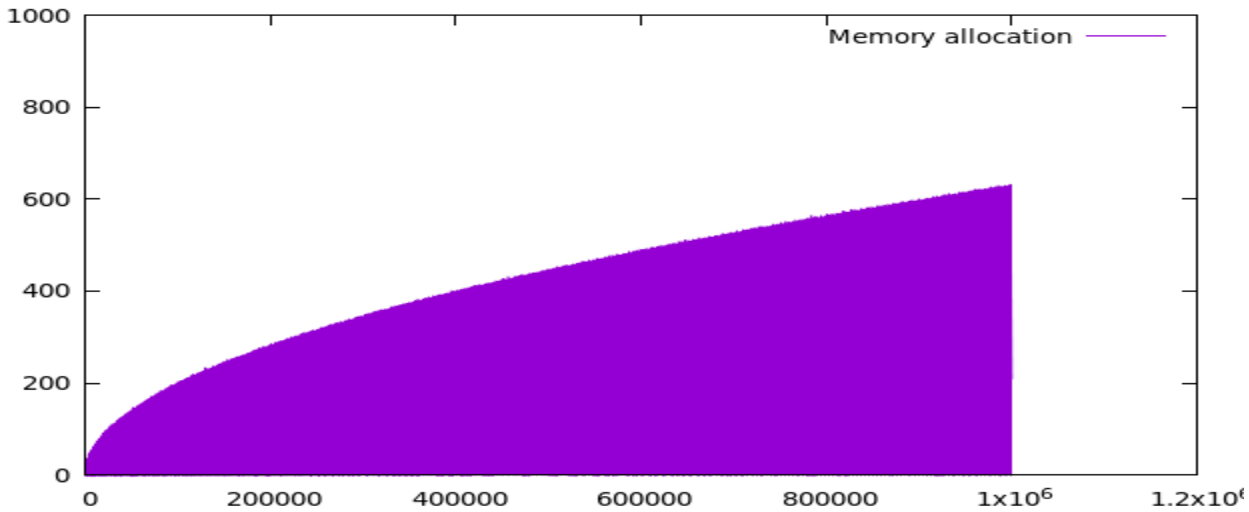
p=0.6


Amortized java
5000
4000
3000
2000
1000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

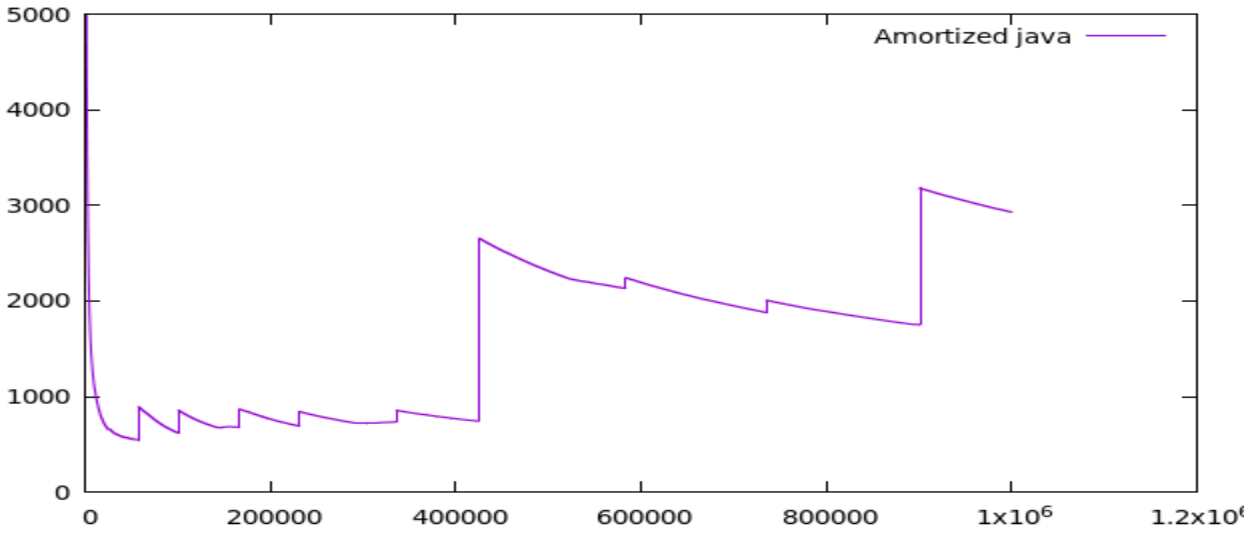

Memory allocation
1000
800
600
400
200
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

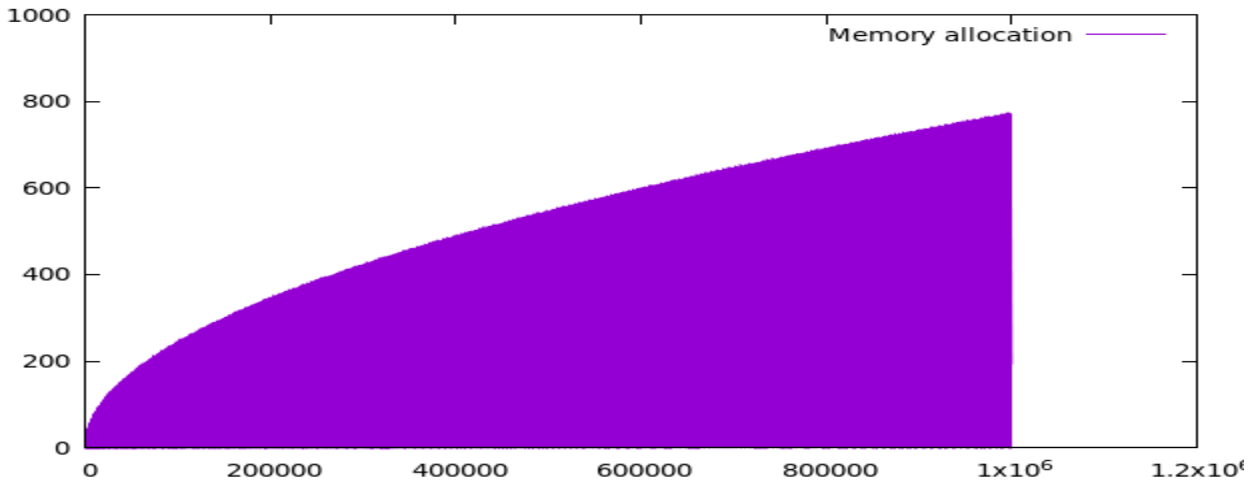
p=0.7


Amortized java
5000
4000
3000
2000
1000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

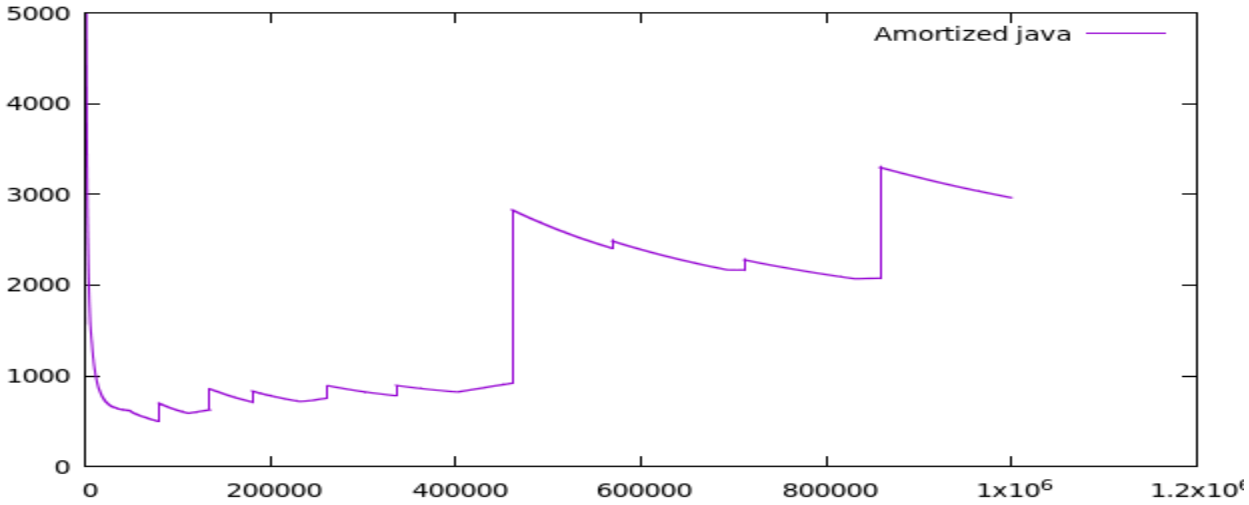

Memory allocation
1000
800
600
400
200
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

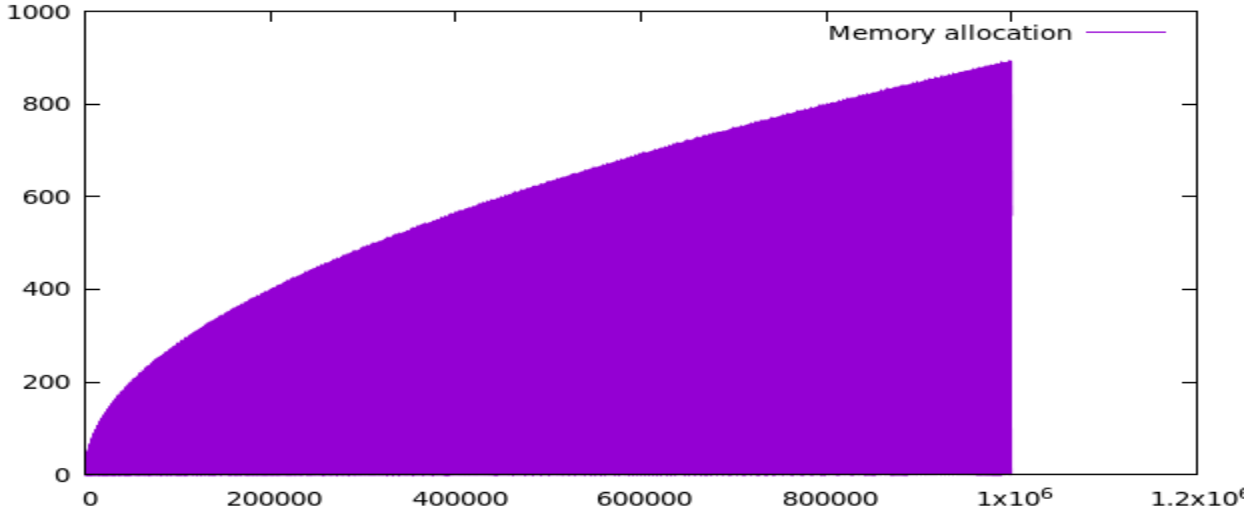
p=0.8


Amortized java
5000
4000
3000
2000
1000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6



p=0.9



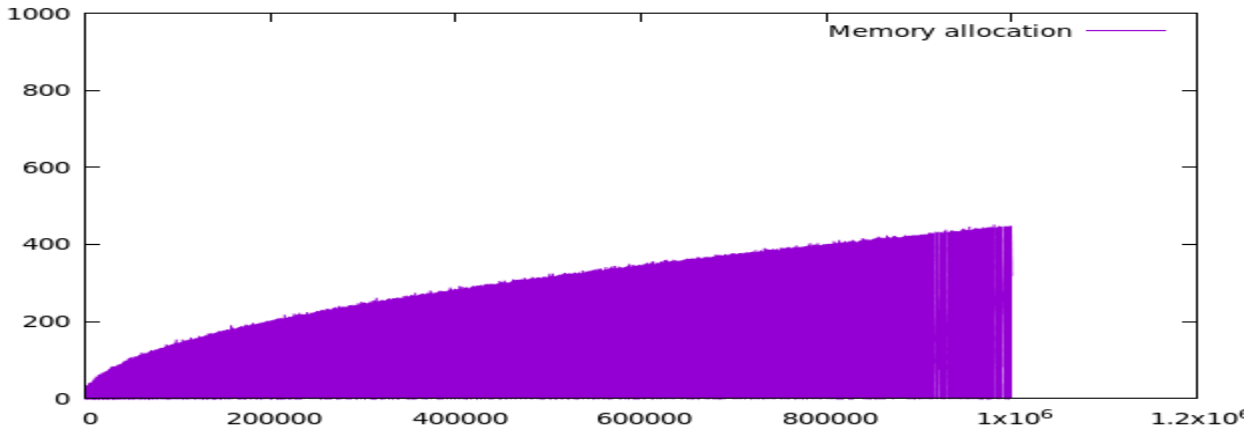


En testant les différentes valeurs de p, la nouvelle stratégie n'améliore pas le coût des opérations ni le gaspillage de mémoire. En dessous et au dessus de p=0.5, on obtient presque les mêmes résultats d'exécution qu'avec la première stratégie, exception faite pour p=0.6.

La valeur de p pour laquelle p semble mieux fonctionner c'est donc p=0.6 car en exécutant plusieurs fois le programme avec ce pourcentage, on constate que le gaspillage de mémoire est moins important dans la deuxième stratégie par rapport à la première.

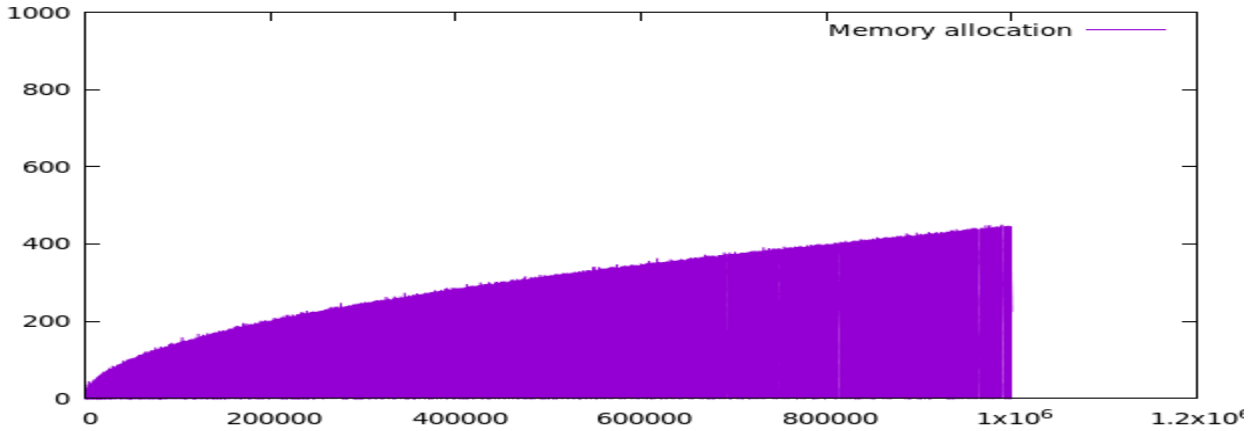
Stratégie n°1

Expérience 1 :



Il reste environ 300 cases mémoires non utilisées.

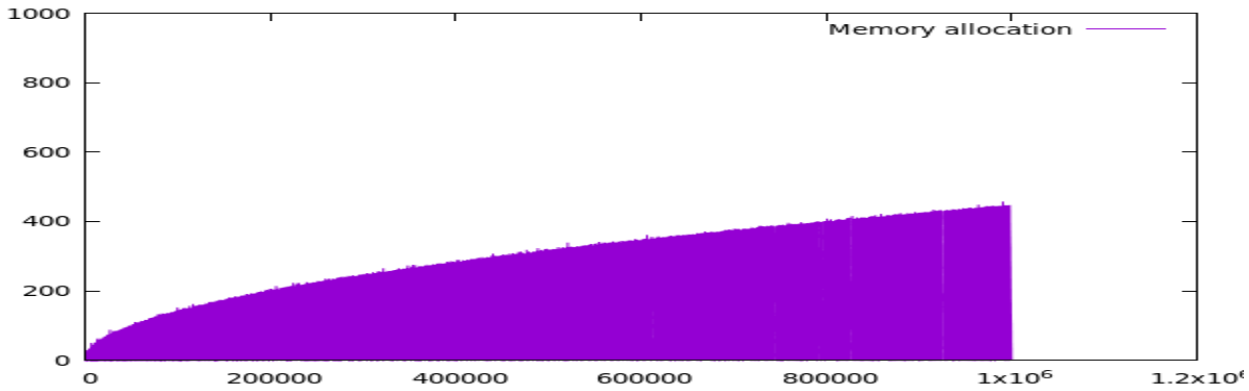
Expérience 2 :



Il reste environ 200 cases mémoires non utilisées.

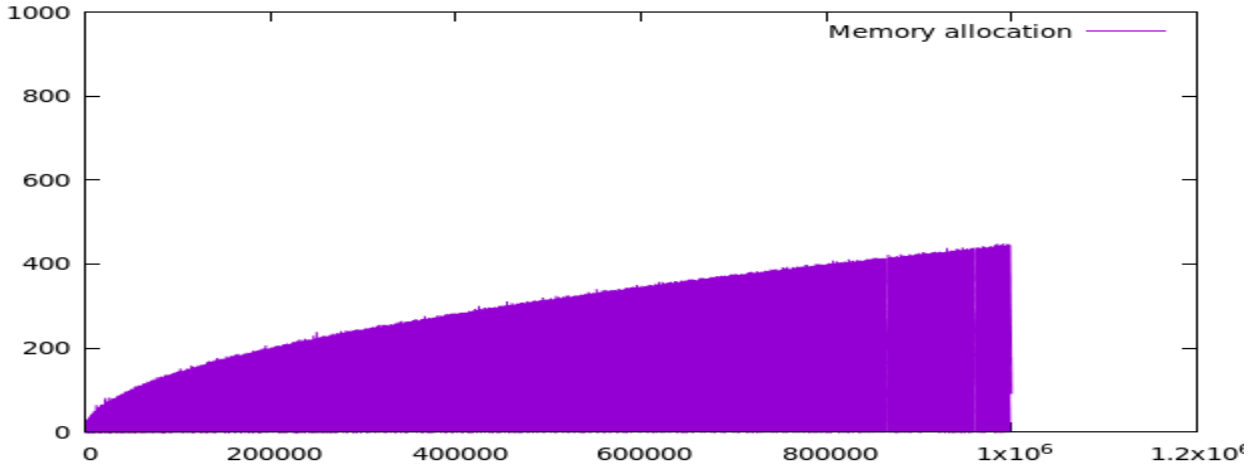
Pour la stratégie n°2

Expérience n°1



Il reste environ 7 cases mémoires non utilisées.

Expérience n°2



Il reste 4 cases mémoires non utilisées.

Synthèse

A travers ce TP, nous avons analysé le coût des opérations quand on fait à la fois des insertions et des suppressions dans une table dynamique.
Nous avons testé 2 stratégie de redimensionnement :

- La première consiste à étendre la table, en majorant sa capacité par la racine carré de cette dernière (soit capacité = capacité + √capacité), quand celle-ci est pleine dans le cas d'une insertion. Dans le cas d'une suppression, elle contracte la table de moitié quand le nombre d'élément dans la table est compris entre la valeur correspondant au quart de la capacité de la table et 4.
- Pour la deuxième stratégie, on a gardé la stratégie précédente en ce qui concerne les insertions. Quant aux suppressions, elle contracte la table, en minorant sa capacité par la racine carré de cette dernière (soit capacité = capacité - √capacité), quand le nombre d'éléments dans la table est inférieur à la moitié de la capacité de la table.

Pour chaque expérience, on a choisi à chaque fois un pourcentage d'insertions et de suppressions nécessaires. Ce pourcentage variant entre 0.1 et 0.9, quand p est inférieur à 0.5, cela signifie qu'il y

a plus d’insertions que de suppressions. En revanche, quand p est supérieur à 0.5, cela signifie qu’il y a plus de suppressions que d’insertions. Et quand p=0.5, cela signifie qu’il y a autant d’insertions que de suppressions.
Suite à de nombreuses expériences, nous constatons que pour un pourcentage donné, la première stratégie est meilleure que la deuxième stratégie, exception faite pour le pourcentage p=0.6.
En définitive, cette première stratégie reste la stratégie a adopté quand on a une suite d’insertions et suppressions.

TP3: TAS BINAIRES ET TAS BINOMIAUX

I – TAS BINAIRES

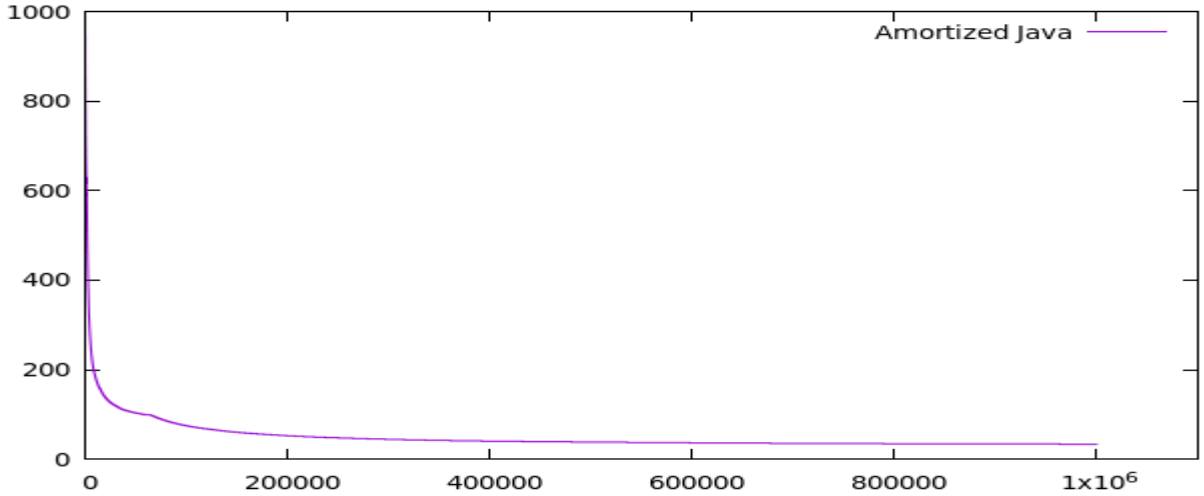
2)Expériences sur l'efficacité en temps et en mémoire d'un tas binaire pour un tas représenté par un tableau de taille fixe

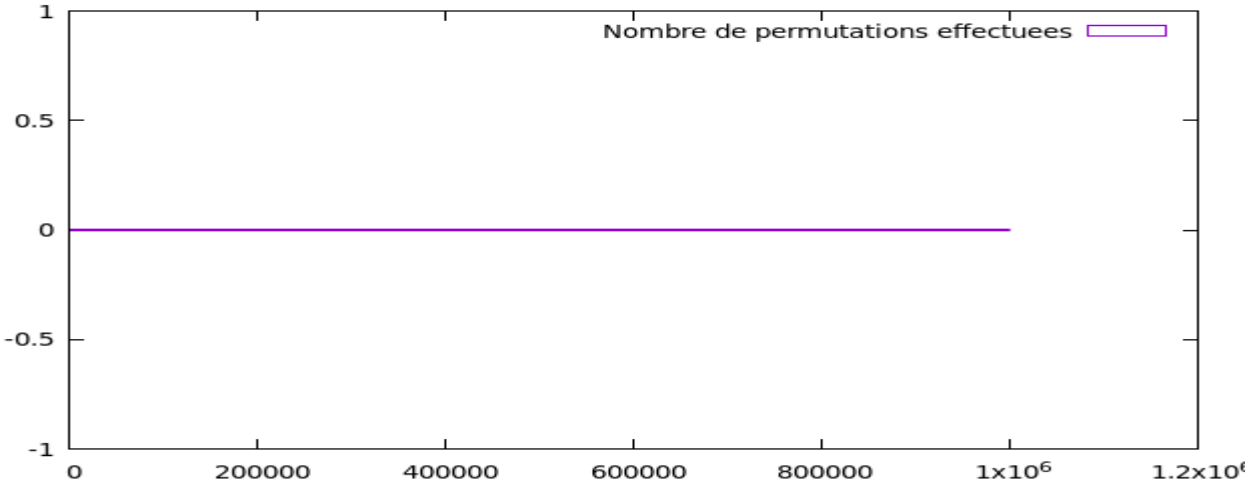
Quand on veut ajouter des éléments dans un tas plein, un message est renvoyé à l'exécution.

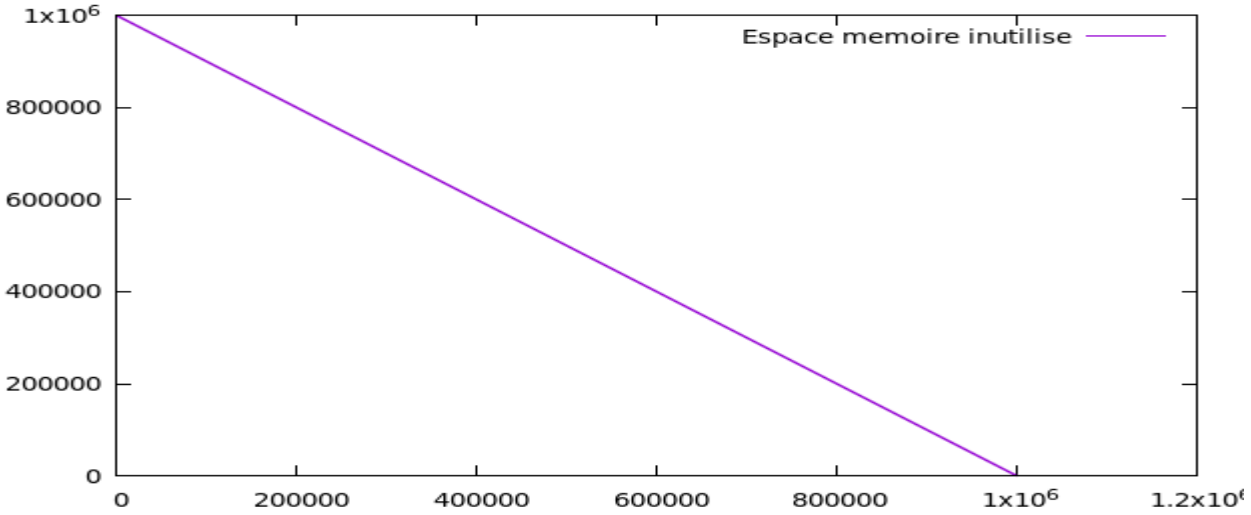
```
(base) yaba@yaba-Latitude-E7240:~/Documents/M1InfoU13/tpSDA/tp3/tasBinaire$ java AjoutCleCroissant
Exception in thread "main" java.lang.RuntimeException: La file est pleine
        at TasBinaire.insert(TasBinaire.java:232)
        at AjoutCleCroissant.main(AjoutCleCroissant.java:33)
```

Cas 1 : On ajoute les clés dans l'ordre croissant

- ***Le nombre d'éléments dans la file est égale à la capacité du tableau représentant le tas***

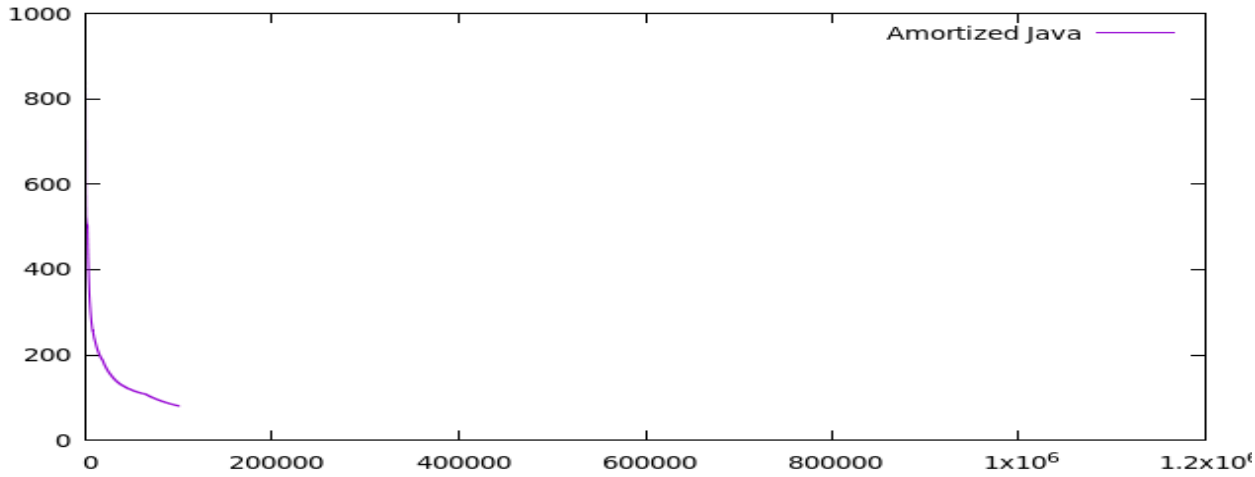


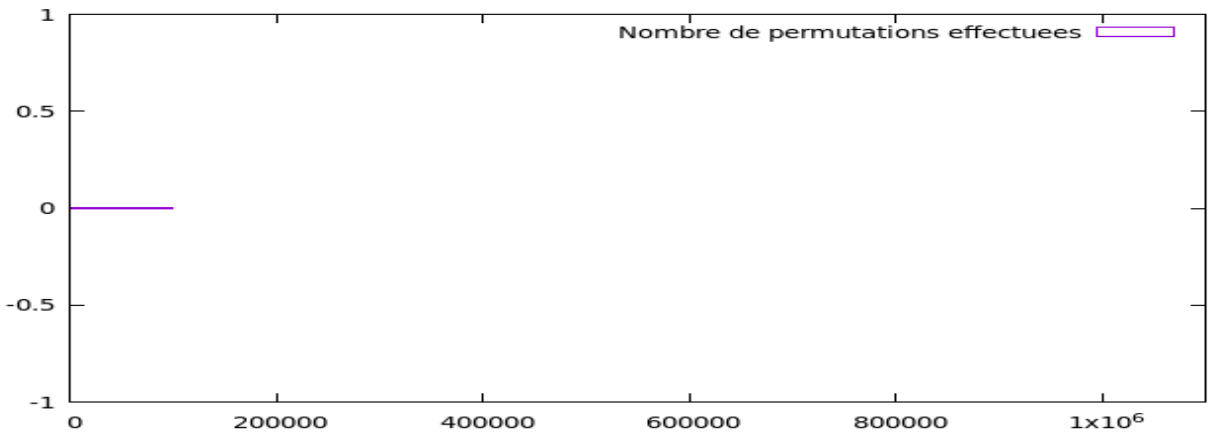


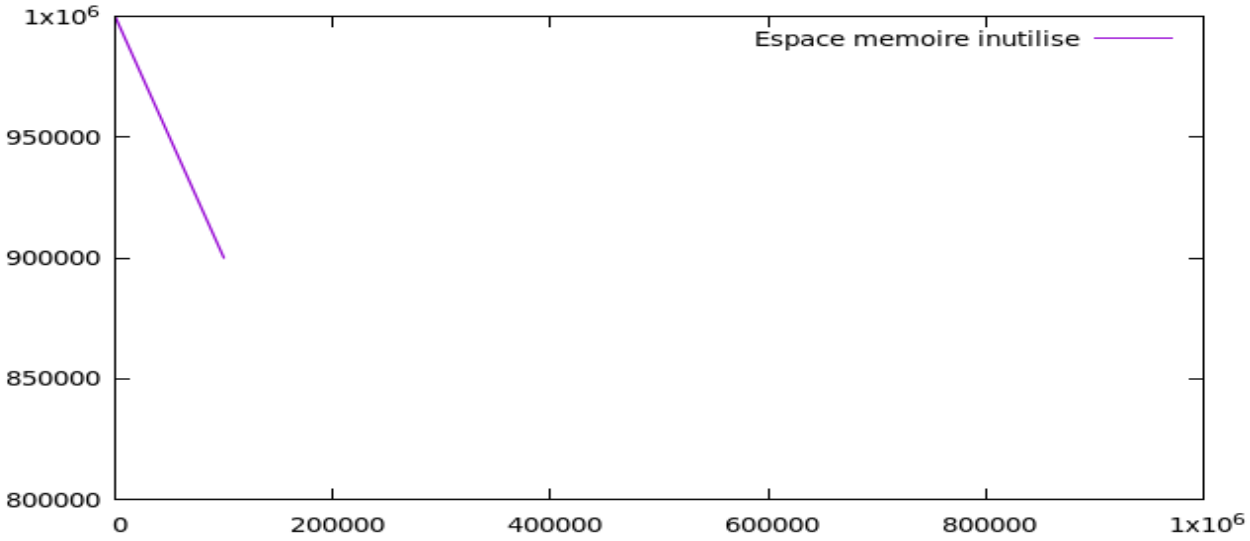


- ***Le nombre d'éléments dans la file est inférieur à la capacité de la file***

Ici, on veut insérer 100000 éléments dans une table de capacité 1000000.

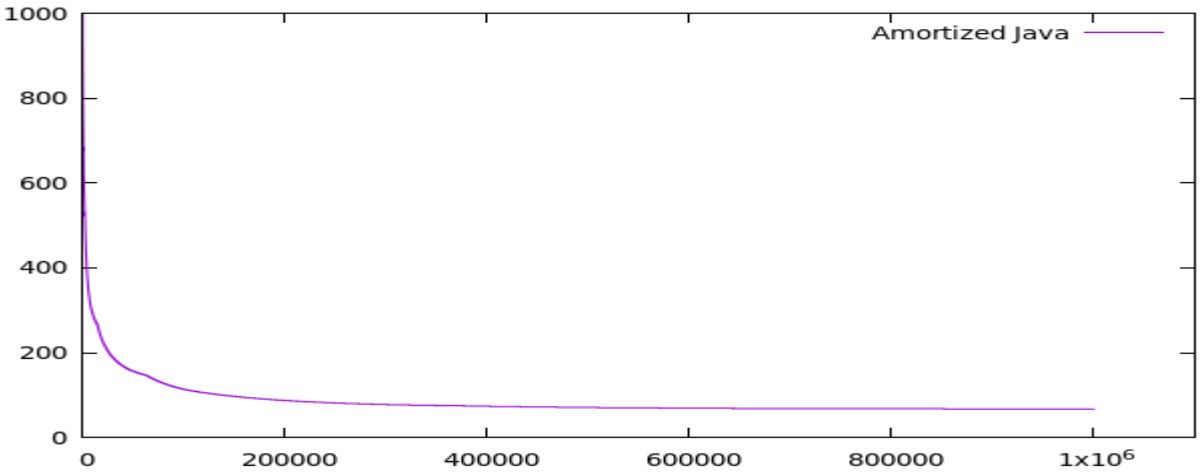


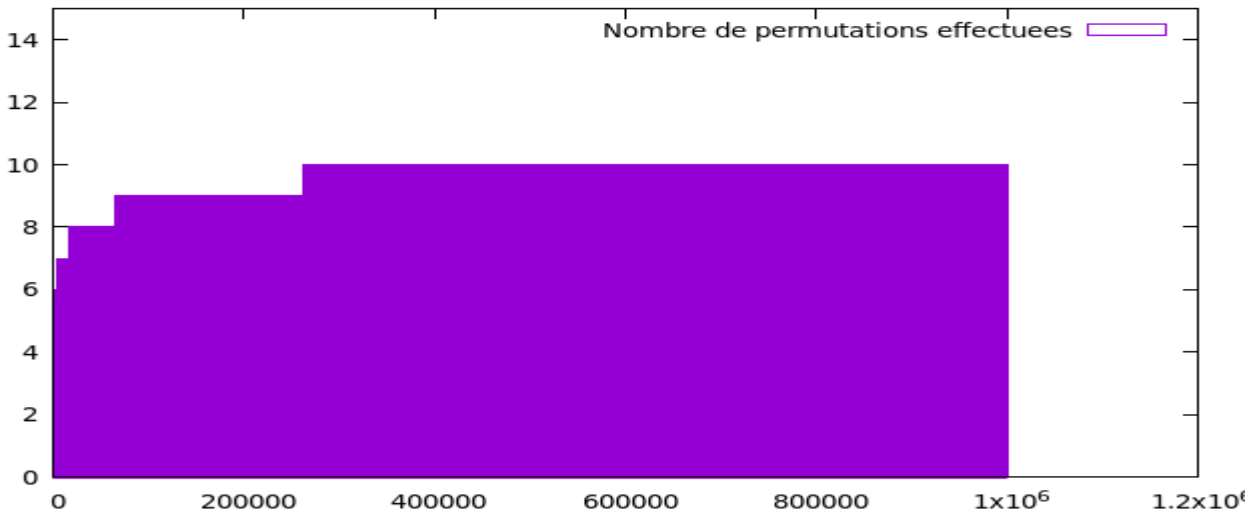


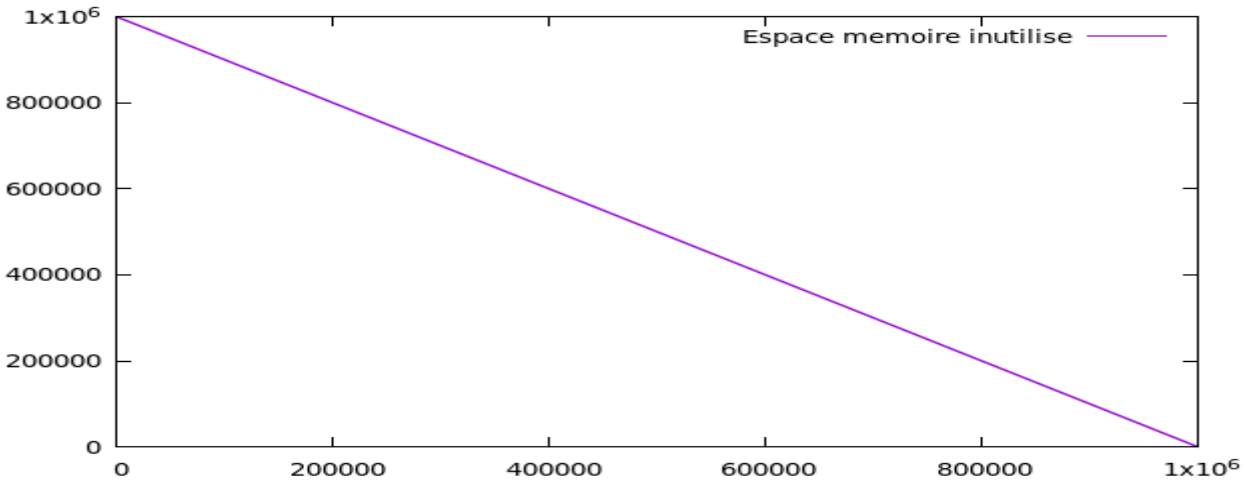


Cas 2 : On ajoute des clés dans l'ordre décroissant

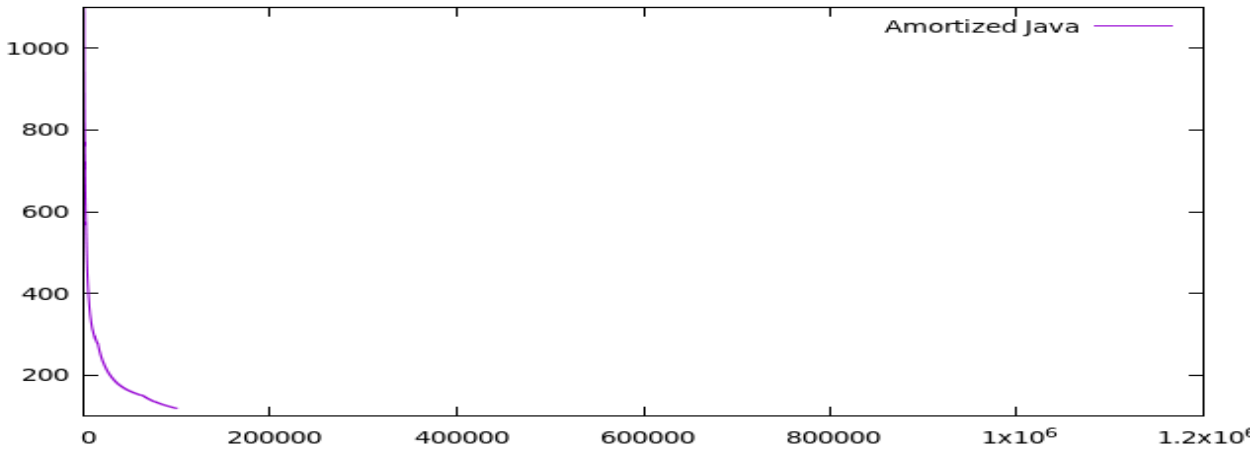
- ***Le nombre d'éléments dans le tas est égale à la capacité du tas***

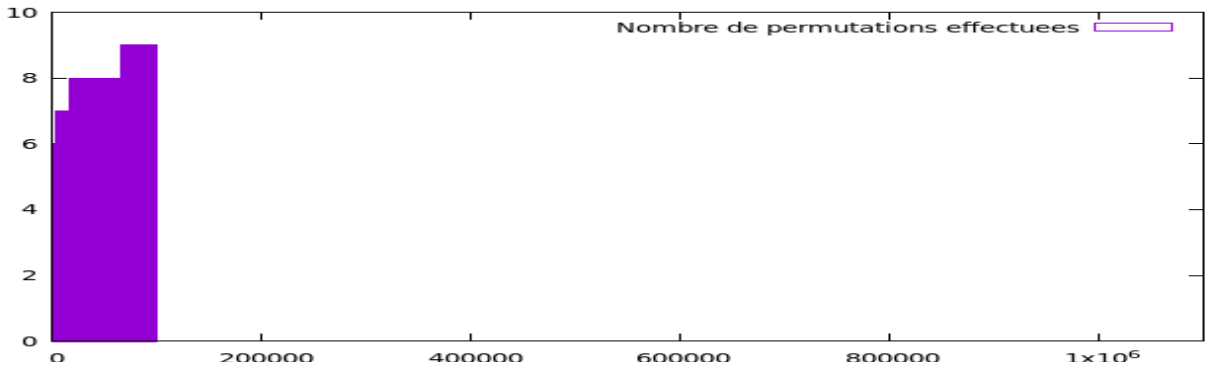


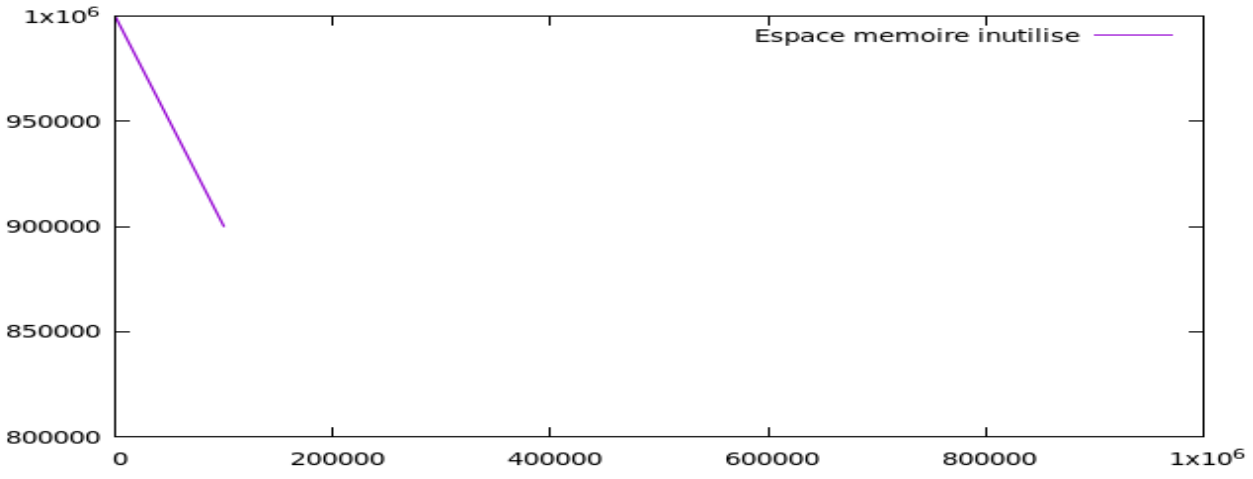




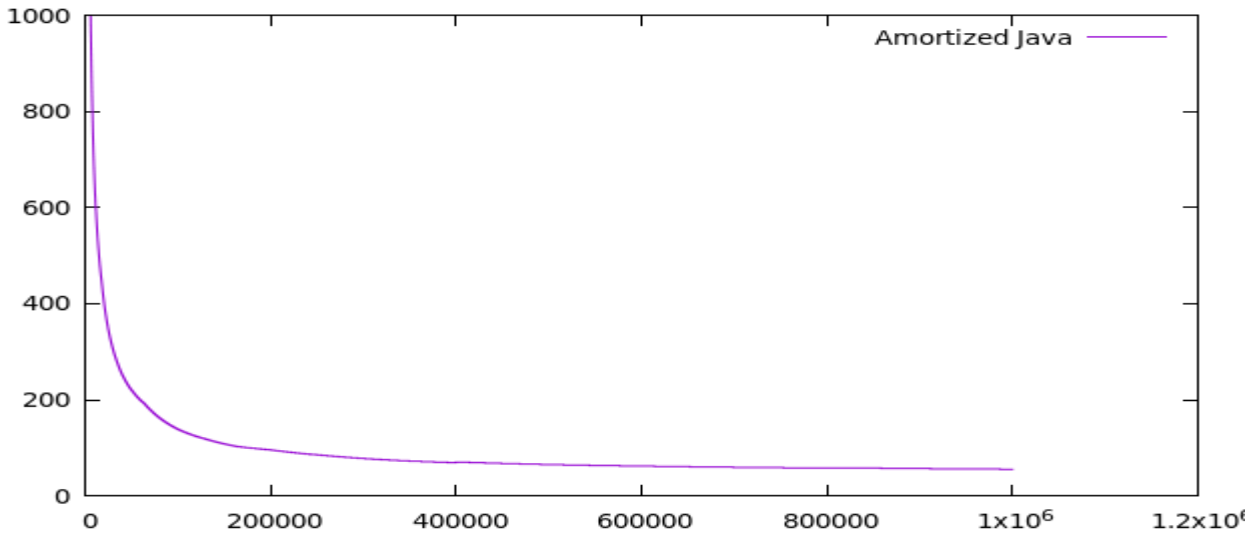
- Le nombre d’éléments dans le tas est inférieur à la capacité du tas

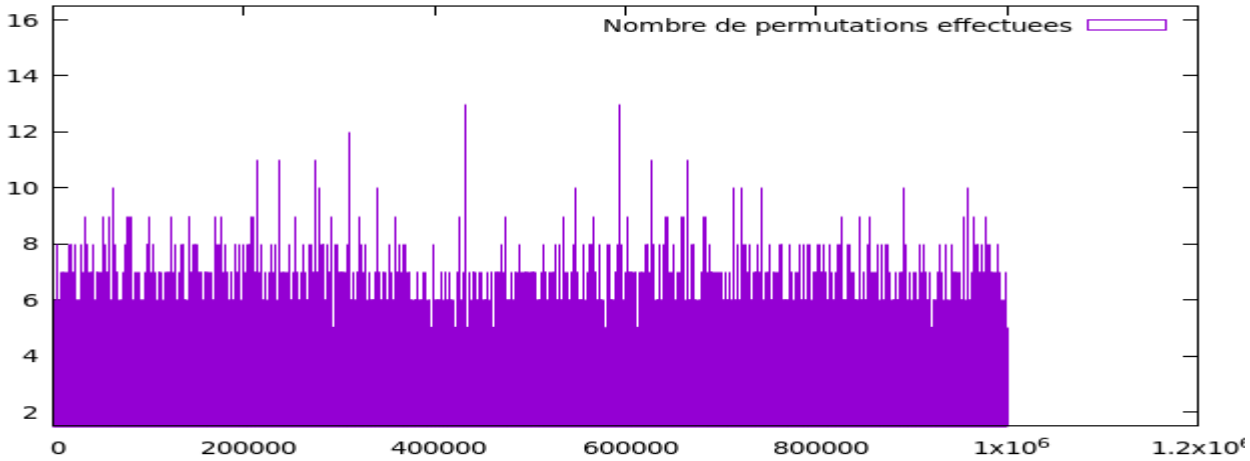


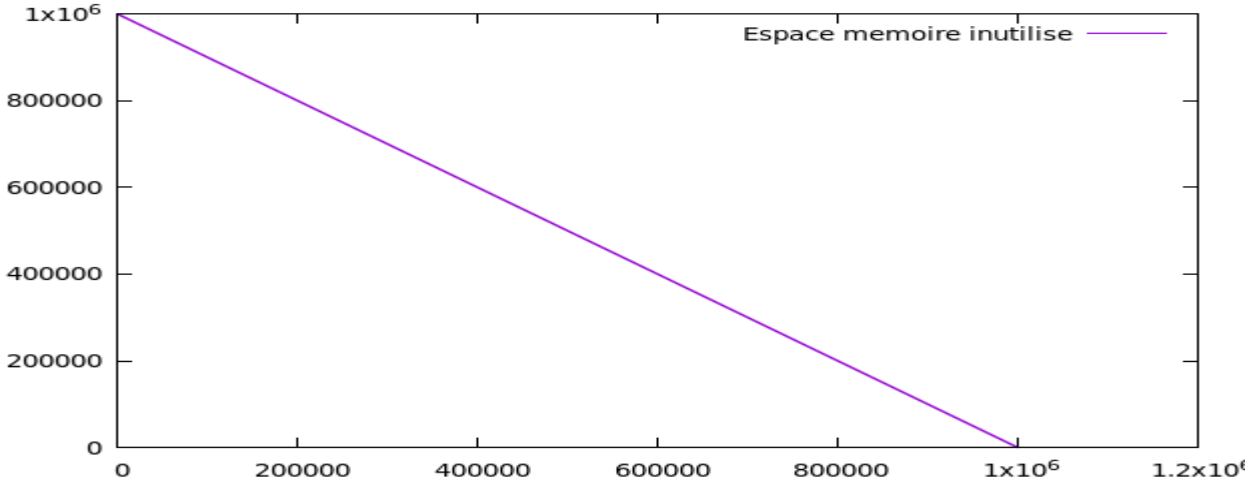




Cas 3 : On ajoute les clés aléatoires

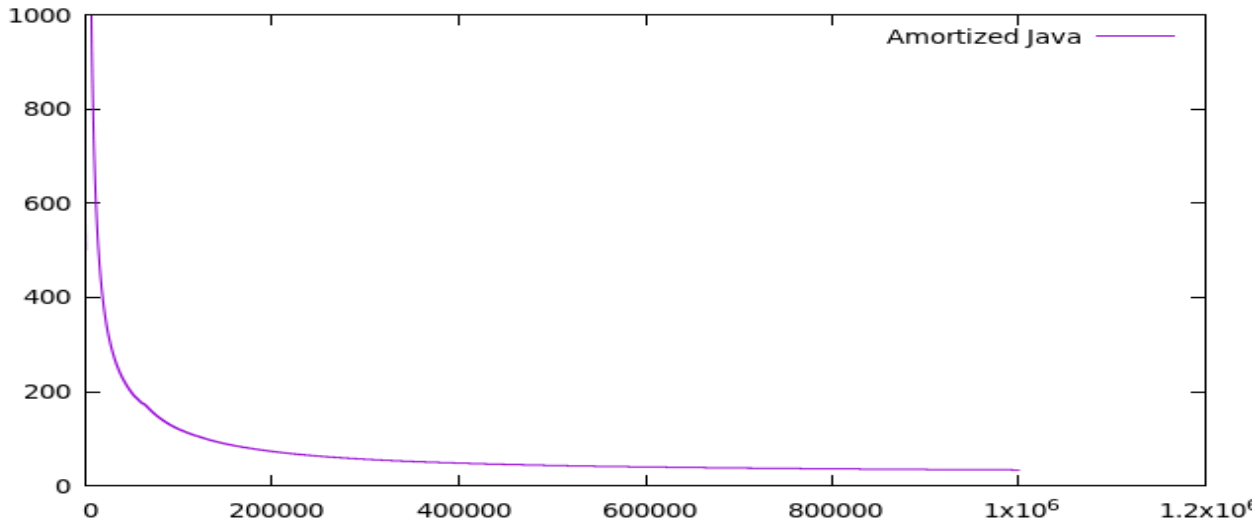


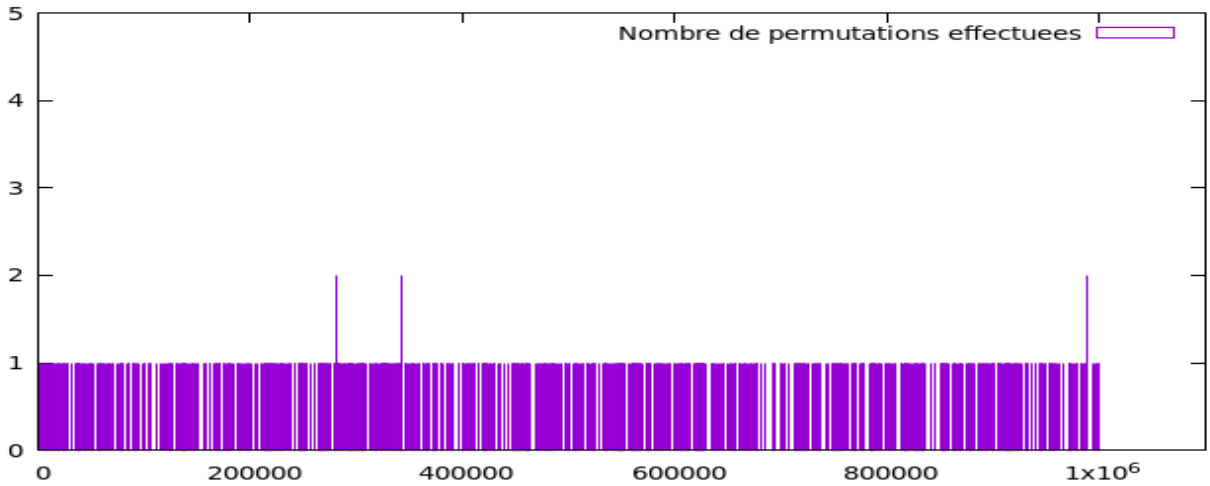


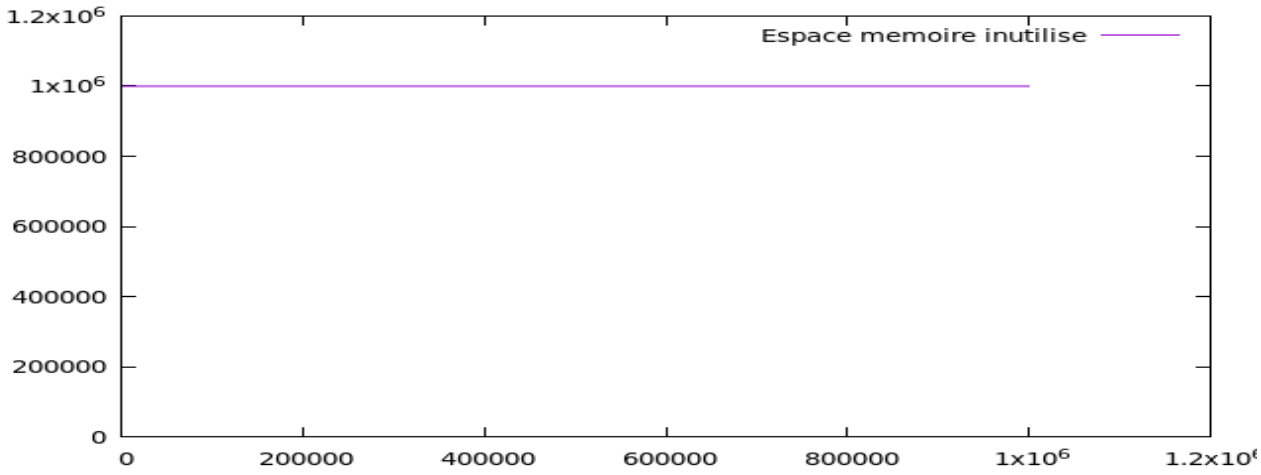


Cas 4 : On ajoute et on extrait les éléments

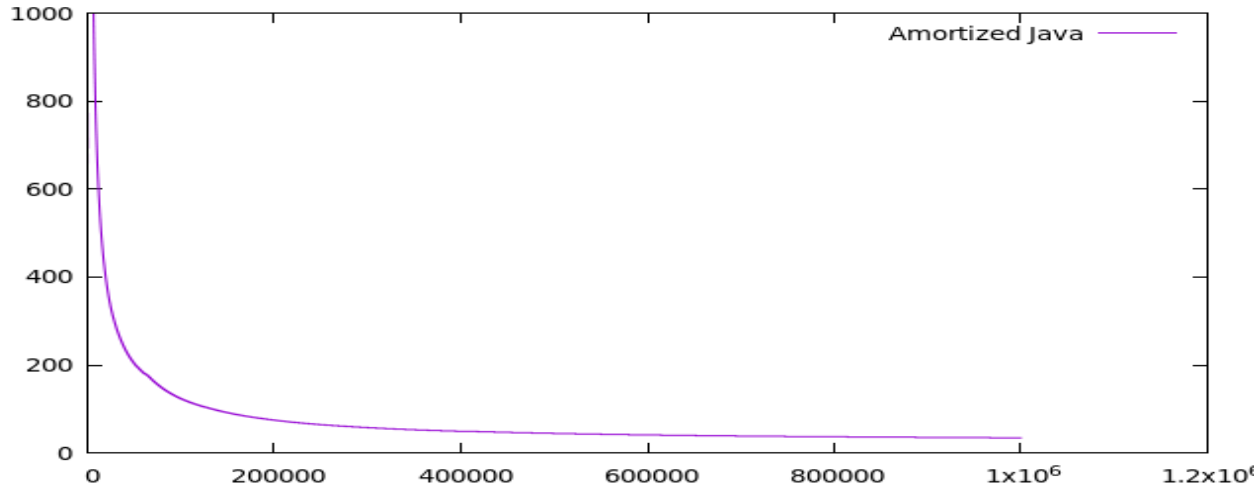
p=0.1

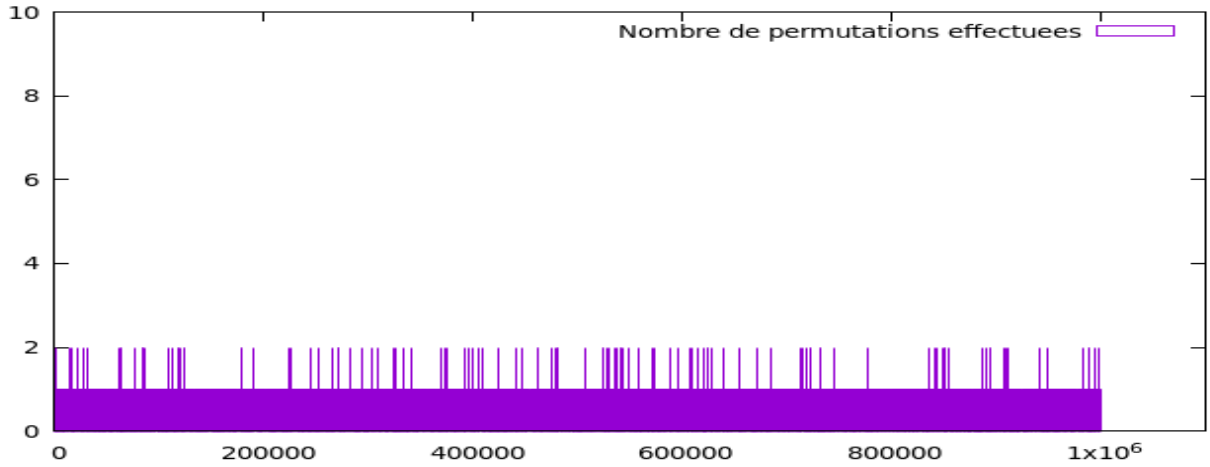


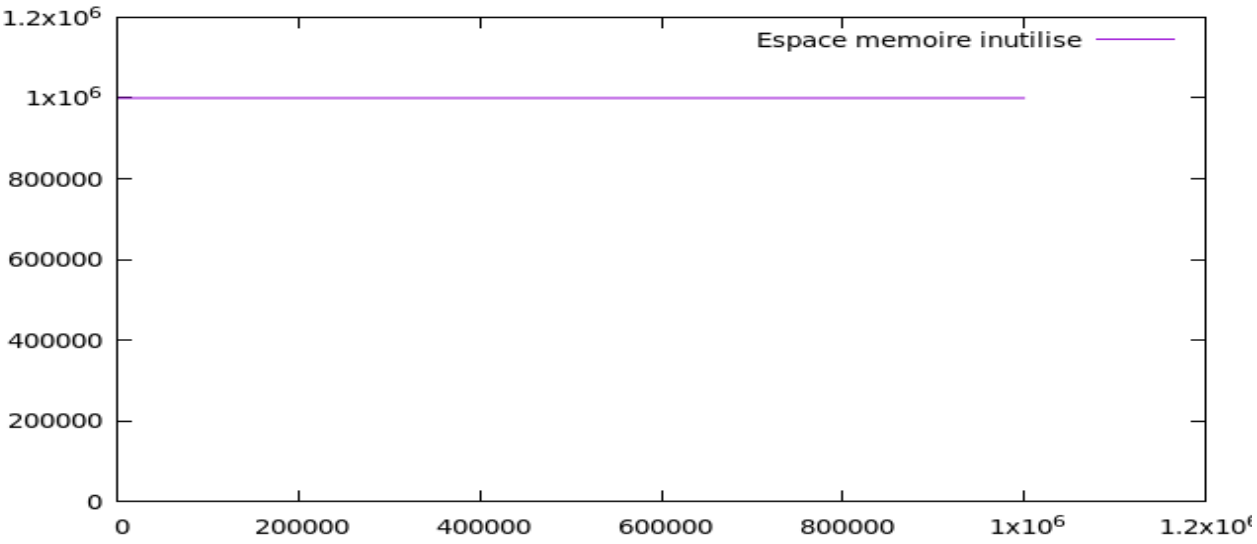




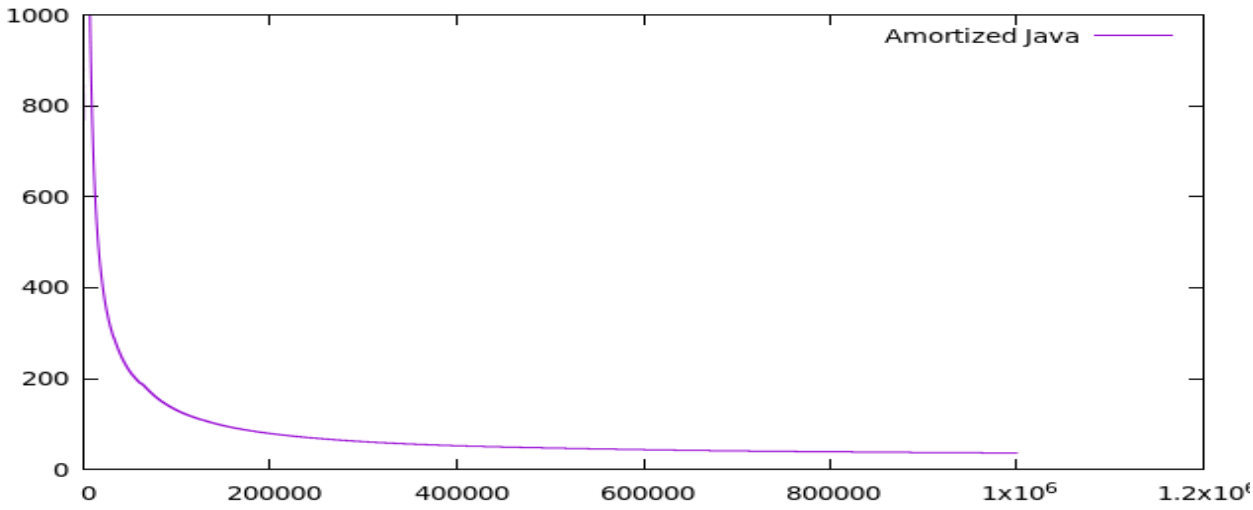
P=0.2

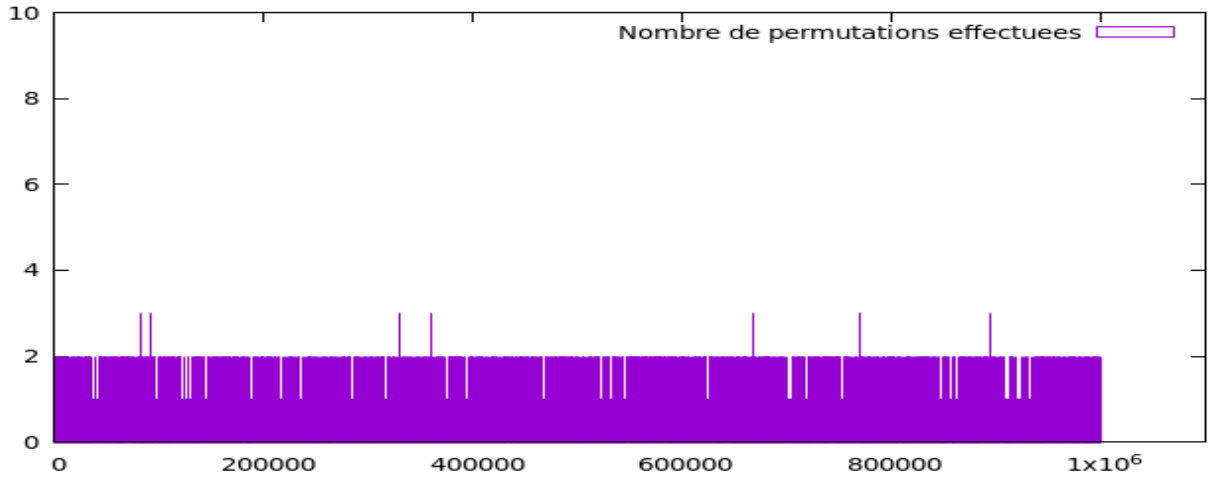

Amortized Java
1000
800
600
400
200
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

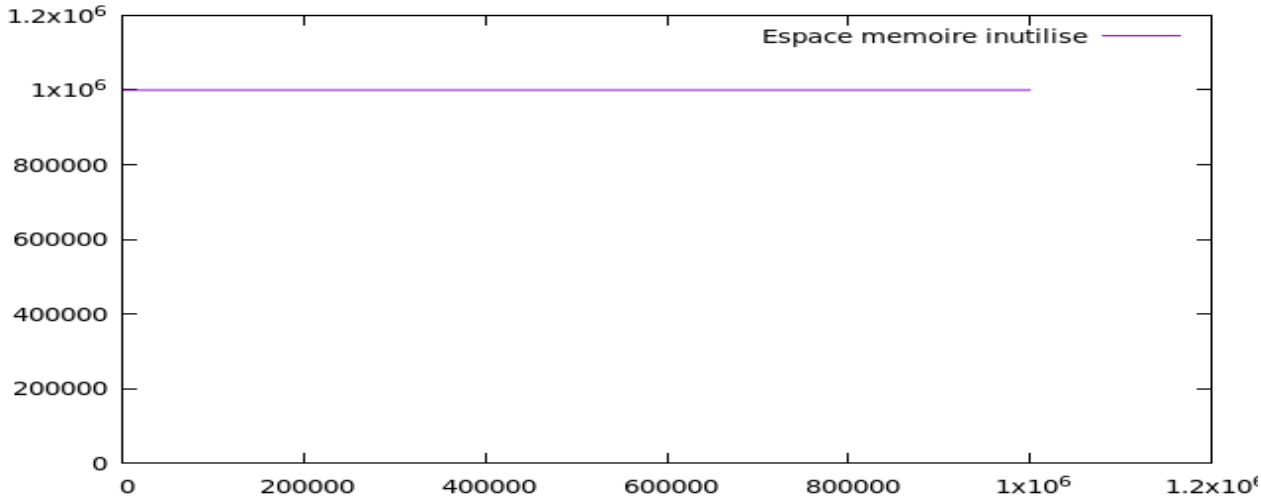

Nombre de permutations effectuees
10
8
6
4
2
0
0
200000
400000
600000
800000
1x10^6


Espace memoire inutilise
1.2x10^6
1x10^6
800000
600000
400000
200000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

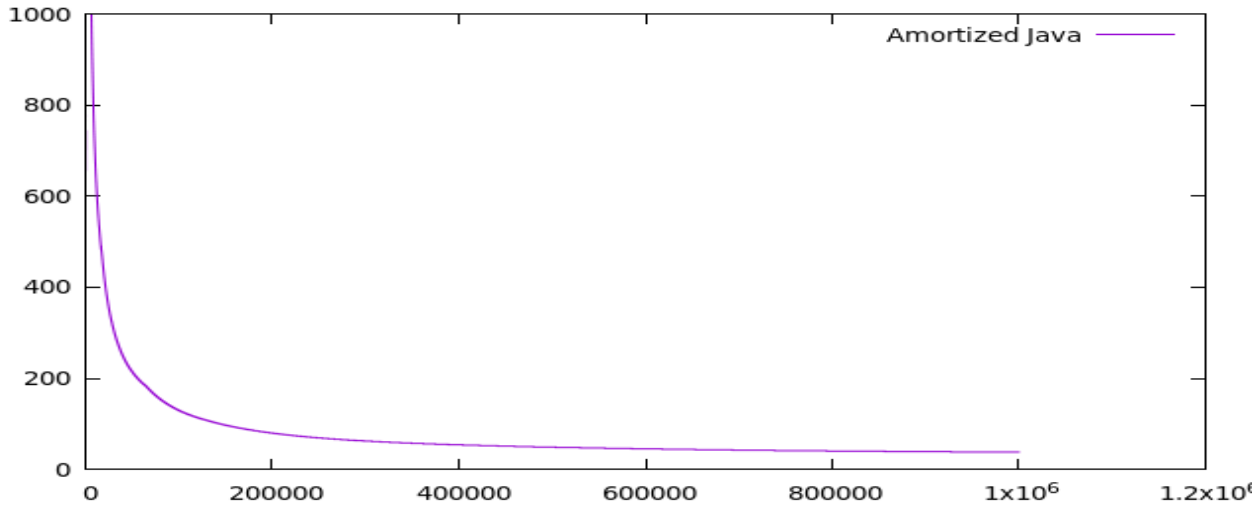
p=0.3

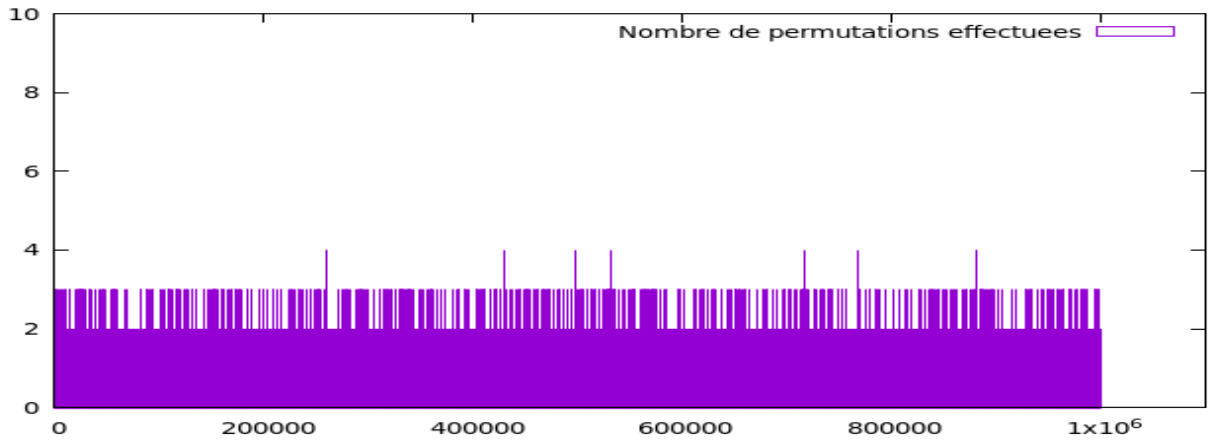

Amortized Java
1000
800
600
400
200
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

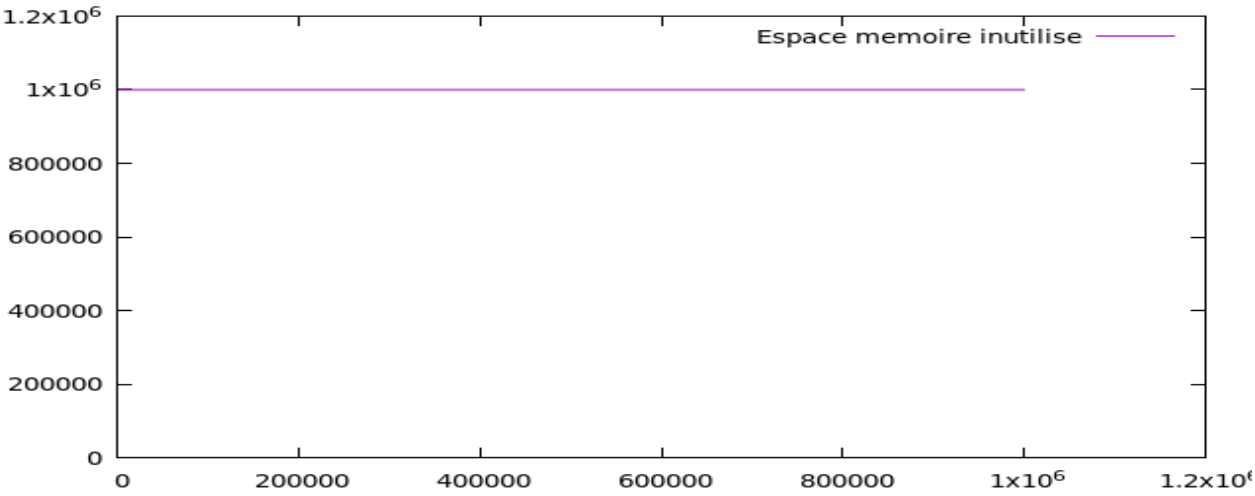

Nombre de permutations effectuees
10
8
6
4
2
0
0
200000
400000
600000
800000
1x10^6


Espace memoire inutilise
1.2x10^6
1x10^6
800000
600000
400000
200000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

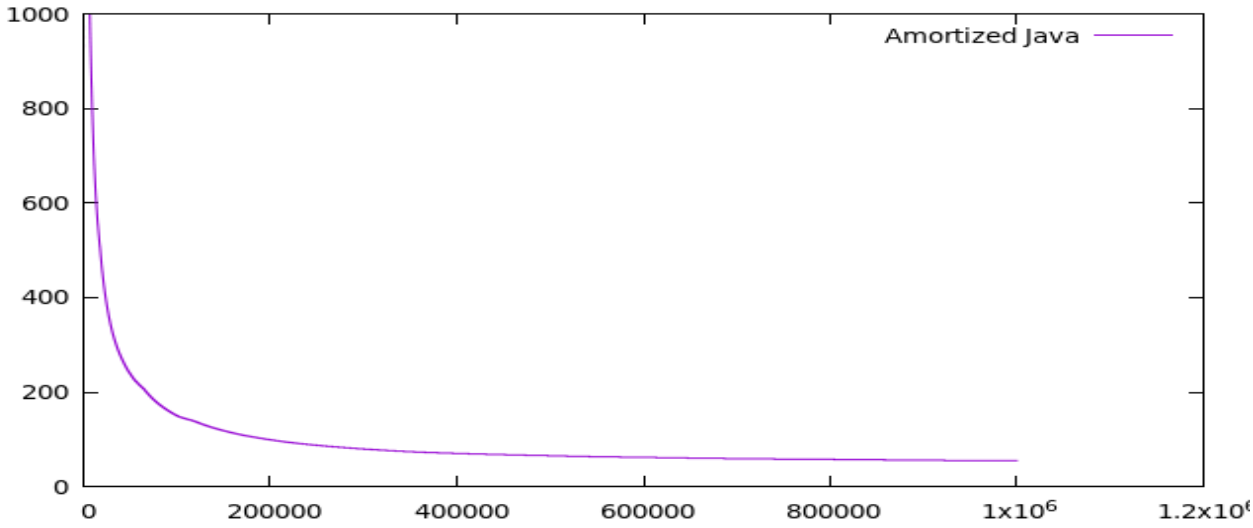
p=0.4

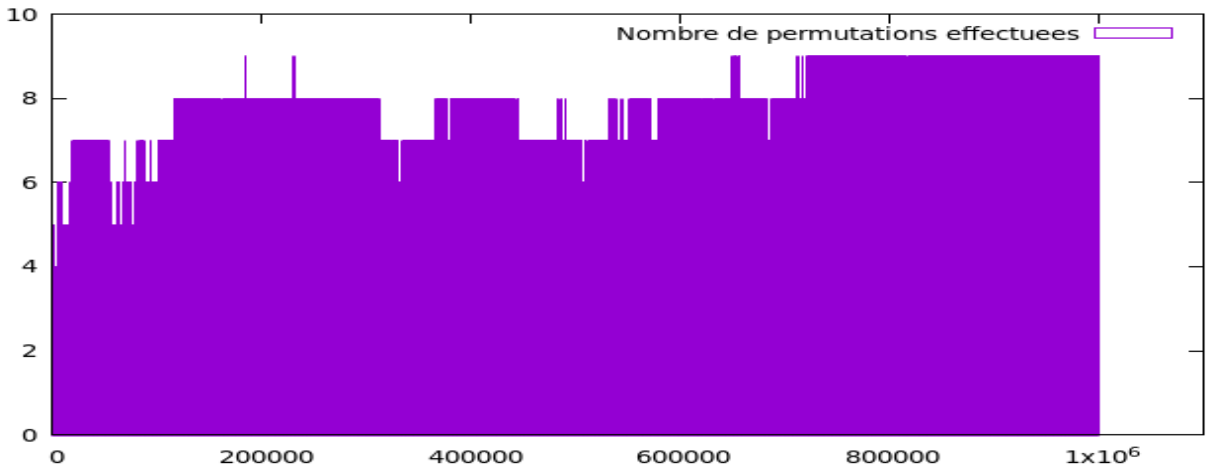

Amortized Java
1000
800
600
400
200
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

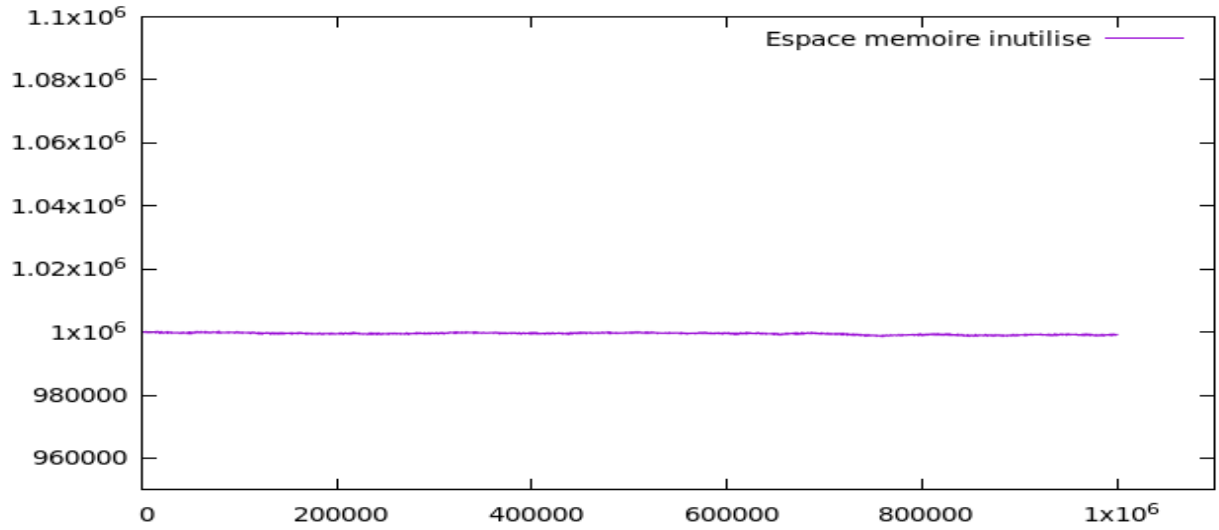

Nombre de permutations effectuees
10
8
6
4
2
0
0
200000
400000
600000
800000
1x10^6


Espace memoire inutilise
1.2x10^6
1x10^6
800000
600000
400000
200000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

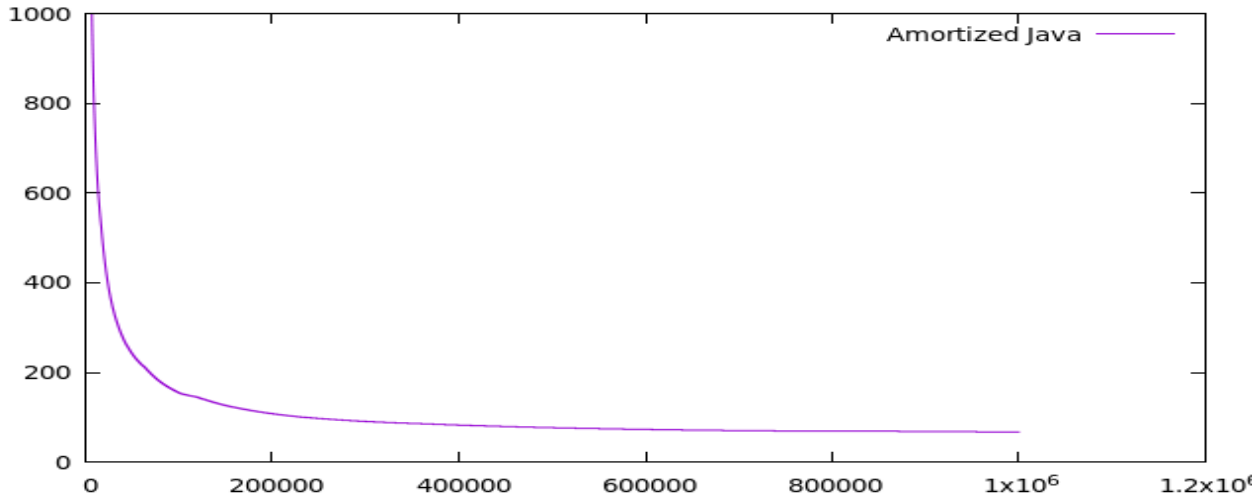
P=0.5

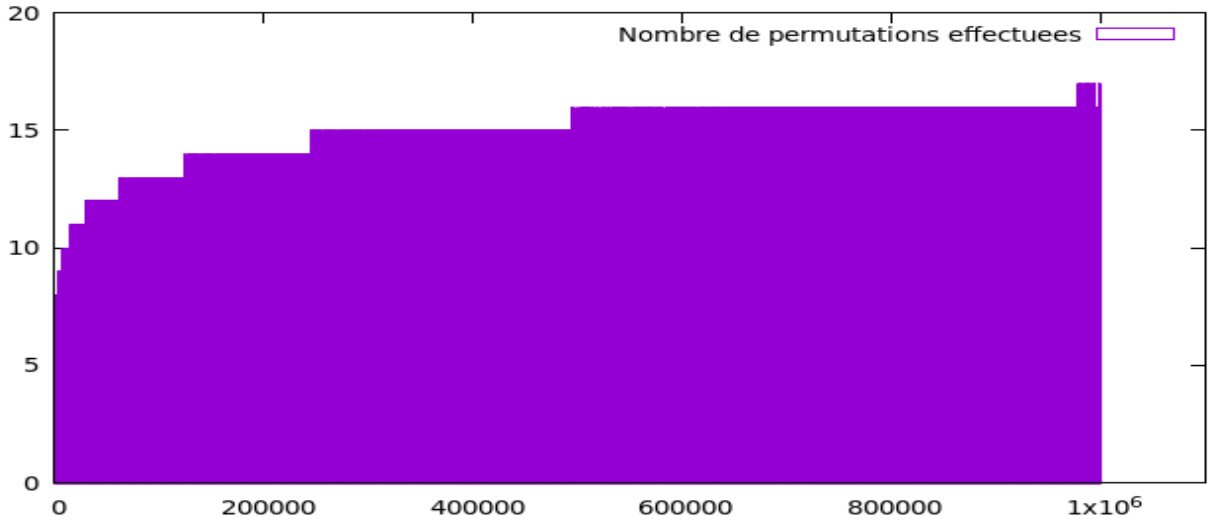

Amortized Java
1000
800
600
400
200
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

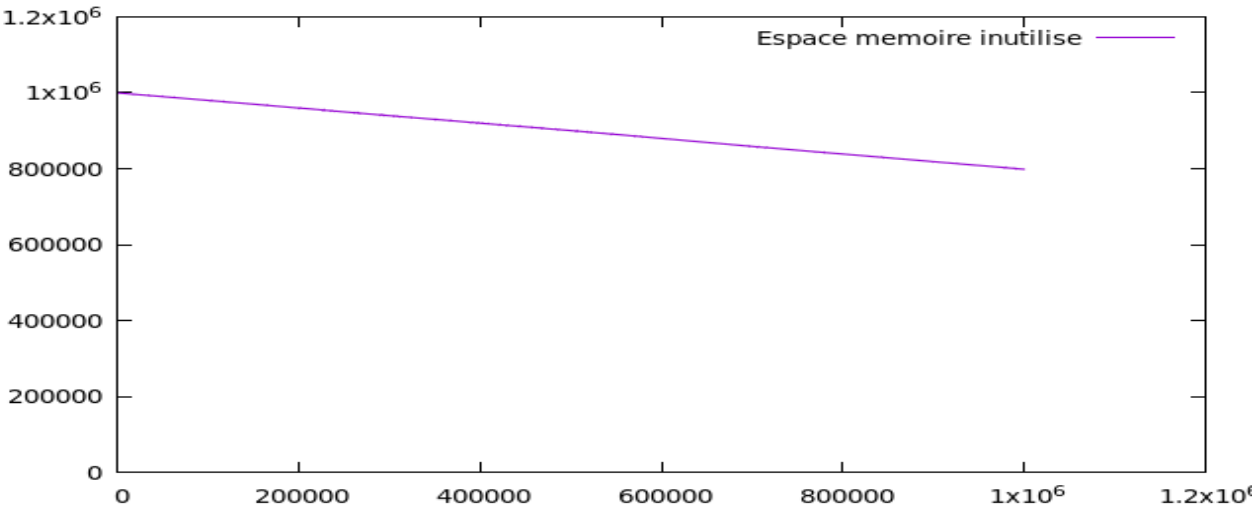

Nombre de permutations effectuees
10
8
6
4
2
0
0
200000
400000
600000
800000
1x10^6


Espace memoire inutilise
1.1x10^6
1.08x10^6
1.06x10^6
1.04x10^6
1.02x10^6
1x10^6
980000
960000
0
200000
400000
600000
800000
1x10^6

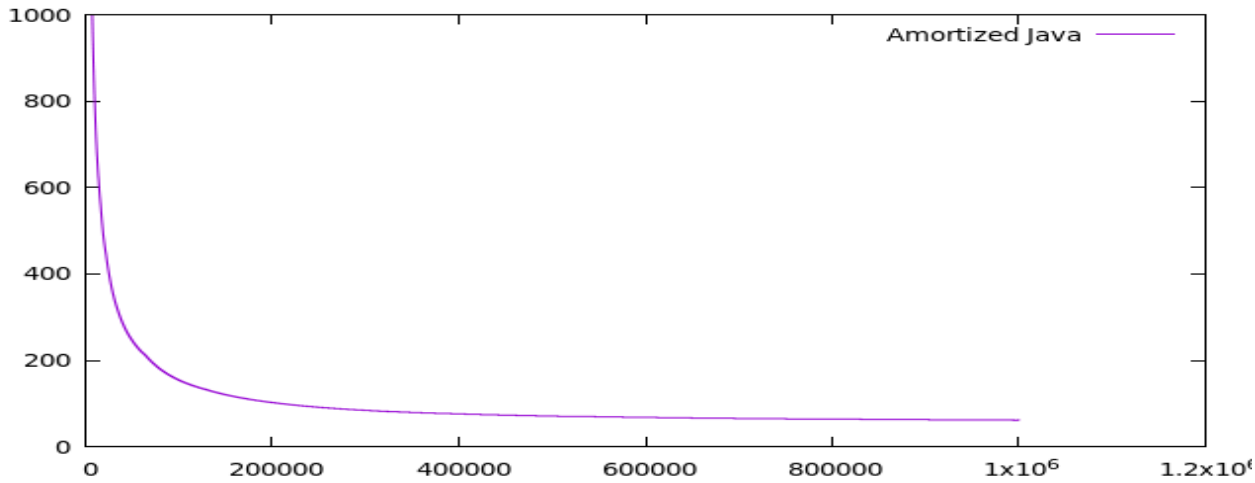
p=0.6

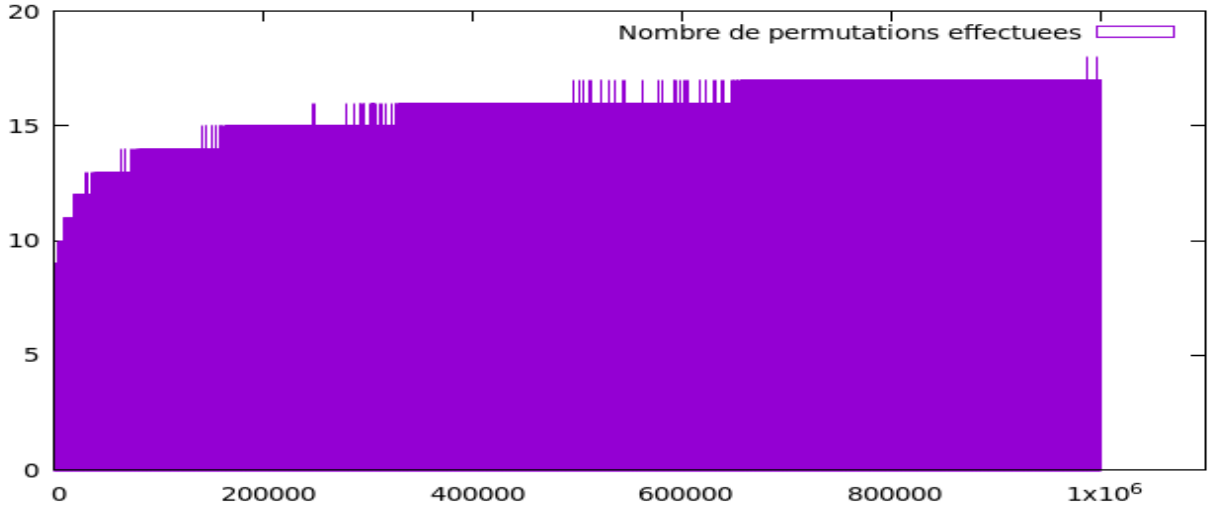

Amortized Java
1000
800
600
400
200
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

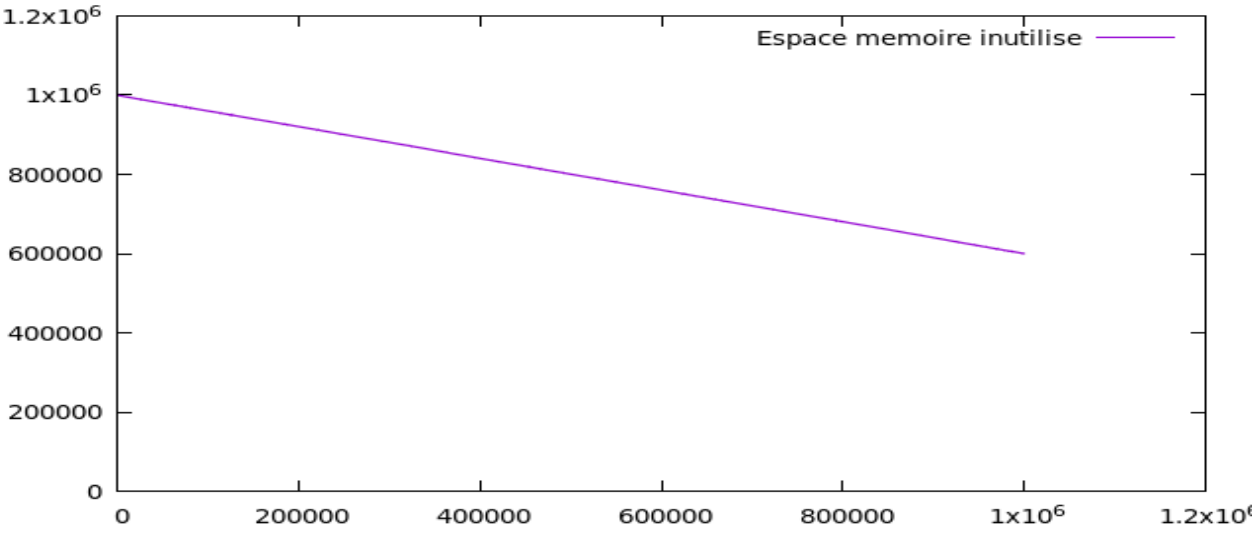

Nombre de permutations effectuees
20
15
10
5
0
0
200000
400000
600000
800000
1x10^6


Espace memoire inutilise
1.2x10^6
1x10^6
800000
600000
400000
200000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

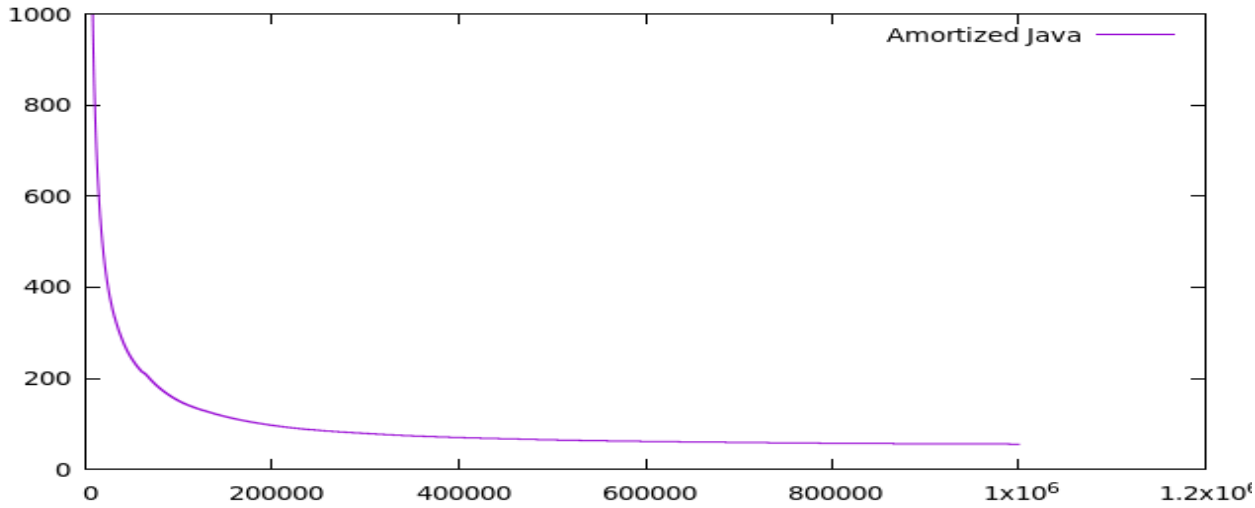
p=0.7

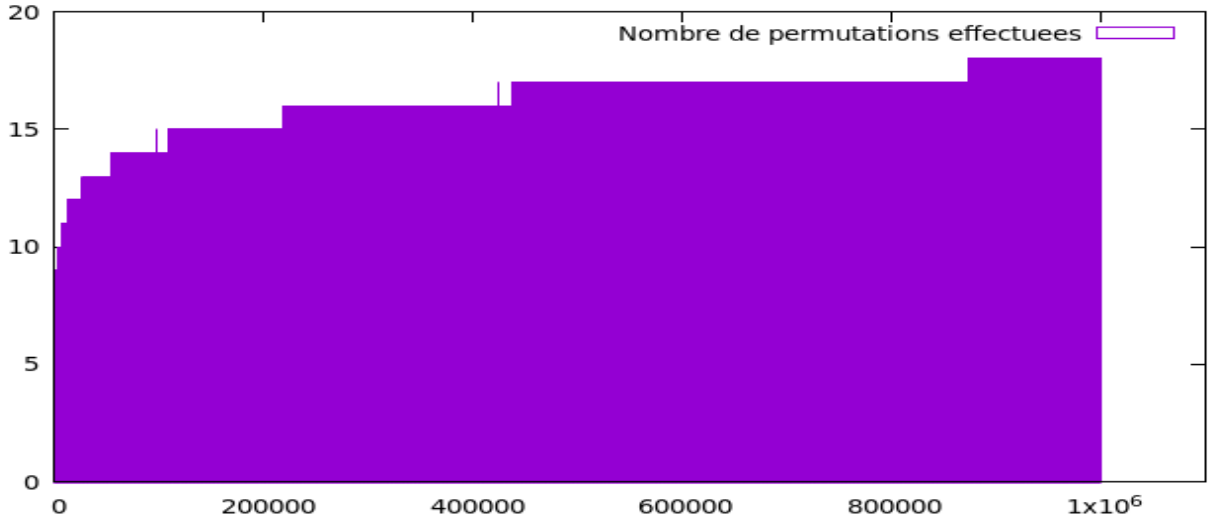

Amortized Java
1000
800
600
400
200
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

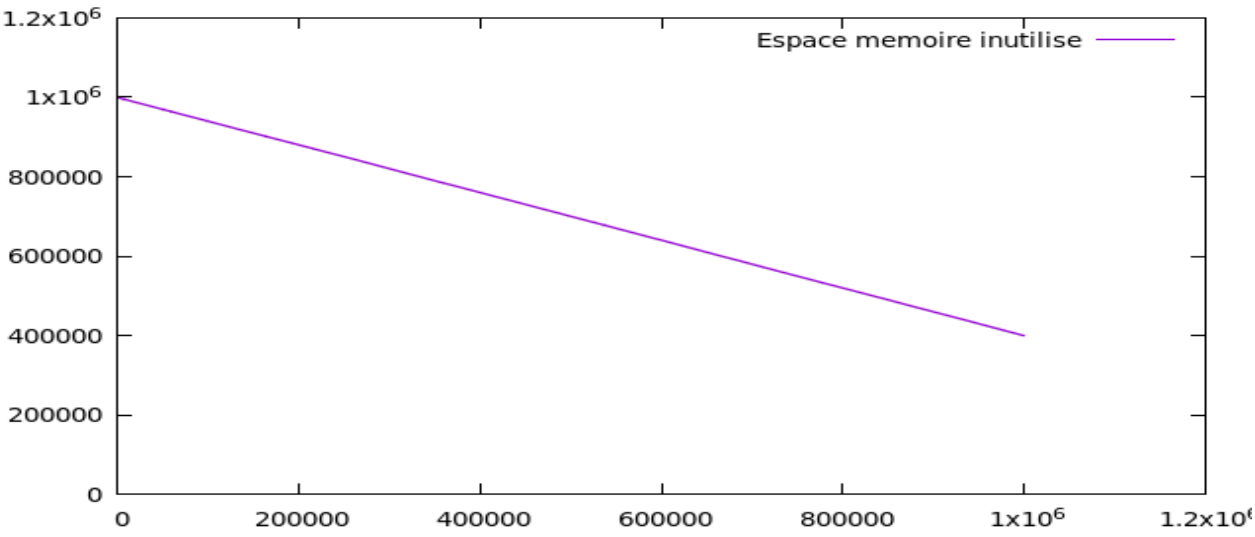

Nombre de permutations effectuees
20
15
10
5
0
0
200000
400000
600000
800000
1x10^6


Espace memoire inutilise
1.2x10^6
1x10^6
800000
600000
400000
200000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

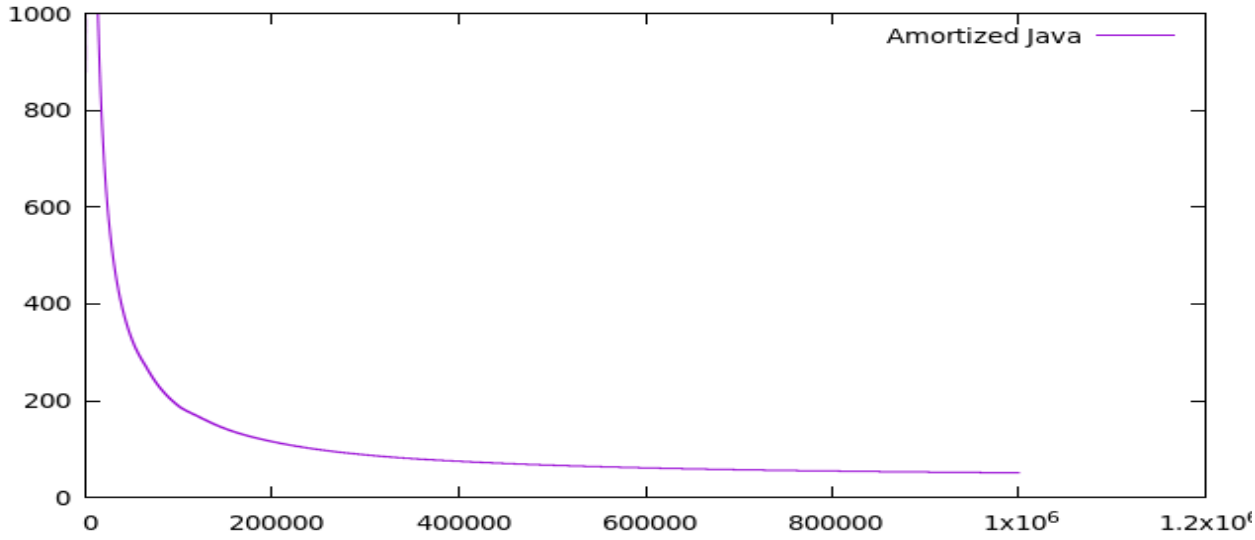
p=0.8

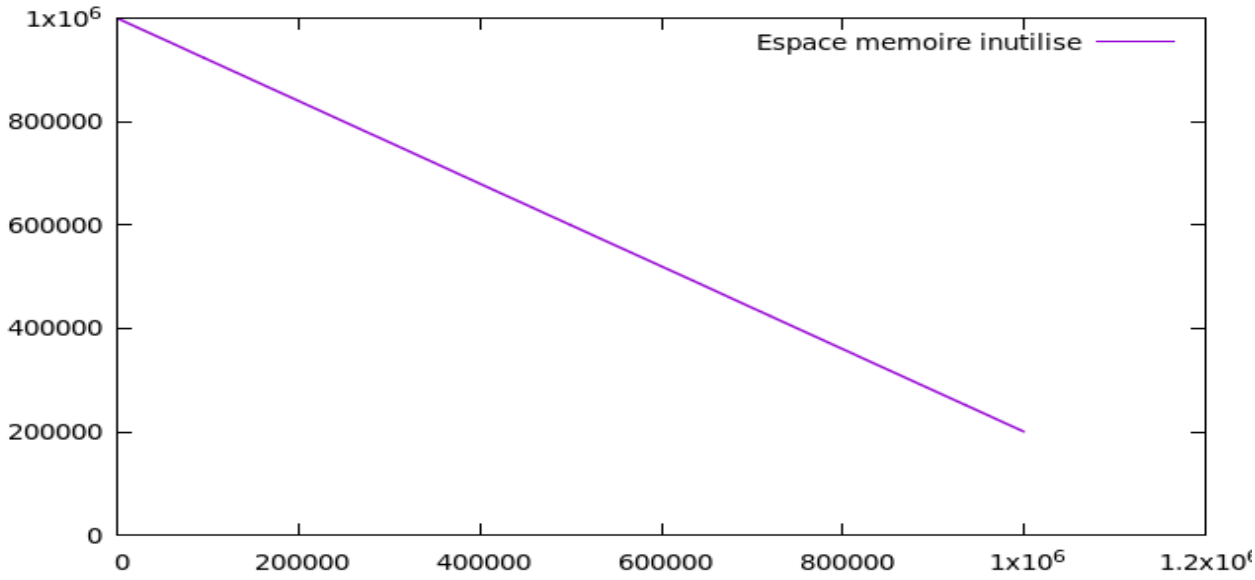

Amortized Java
1000
800
600
400
200
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6


Nombre de permutations effectuees
20
15
10
5
0
0
200000
400000
600000
800000
1x10^6


Espace memoire inutilise
1.2x10^6
1x10^6
800000
600000
400000
200000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

p=0.9





Analyse

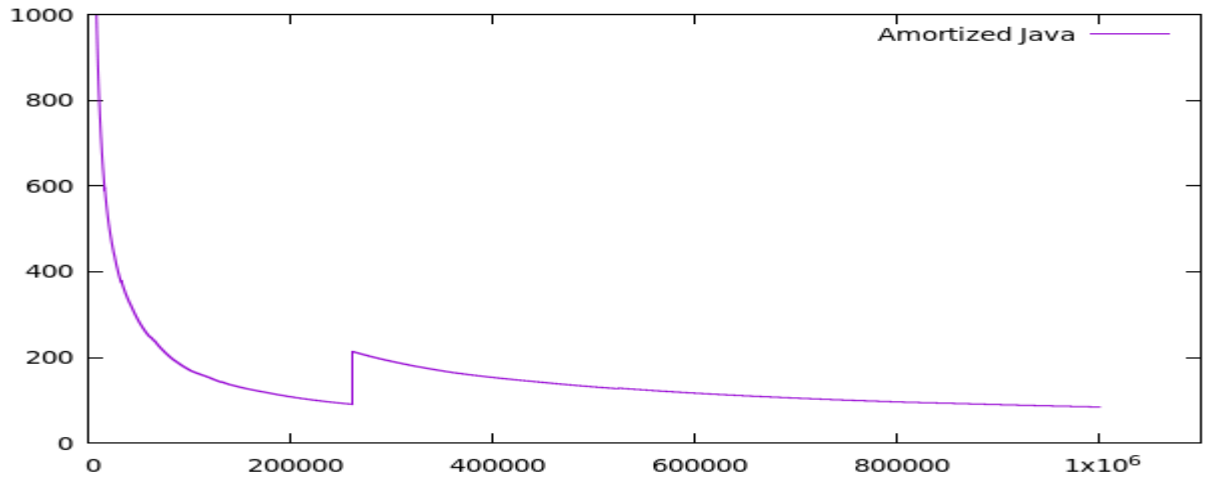
Dans le cas où le tas binaire est représenté par un tableau de taille fixe, le coût amorti est élevé en début pour compenser les opérations ultérieures.

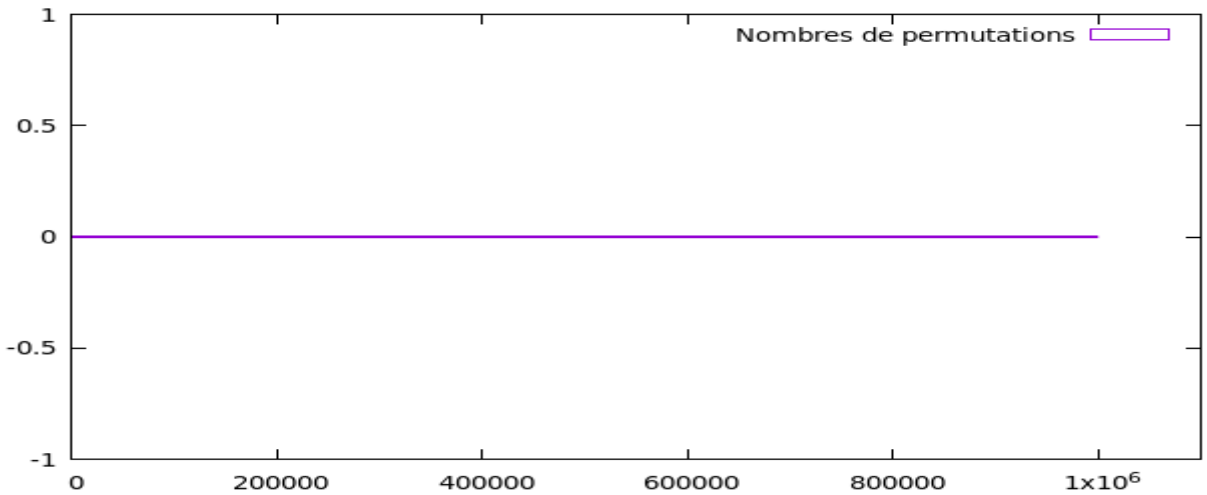
- **Quand les clés sont ajoutées dans l'ordre croissant**, le coût amorti diminue le plus bas possible (soit environ 25) pour les opérations ultérieures. Il n'y a pas de permutation et pas de gaspillage de mémoire, sauf lorsque le nombre d'éléments ajoutés est inférieur à la capacité du tableau.
- **Quand les clés sont ajoutées dans l'ordre décroissant**, le coût amorti baisse d'environ 50 pour les opérations ultérieures, un peu moins que dans le cas où les clés sont croissants. Il y a des permutations qui peuvent varier de 0 à 10. En ce qui concerne l'espace mémoire inutilisée, il n' y a pas de gaspillage de mémoire, sauf lorsque le sauf lorsque le nombre d'éléments ajoutés est inférieur à la capacité du tableau.

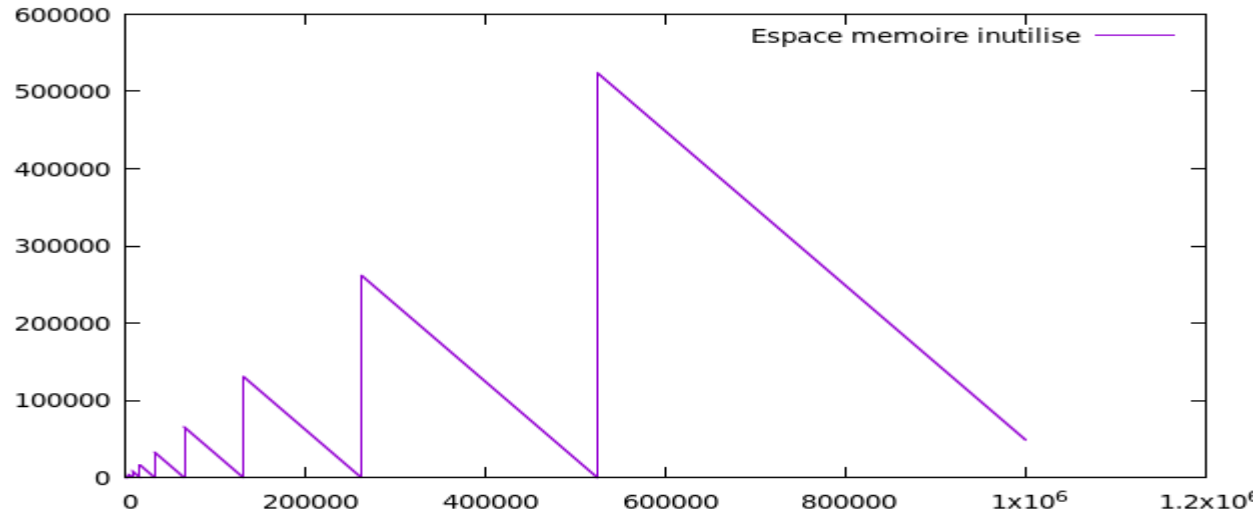
- **Quand les clés sont ajoutées de façon aléatoire**, le coût amorti baisse d'environ 70 pour les opérations ultérieures, un peu moins que dans le cas où les clés sont croissants. Les permutations dépendent des valeurs tirés aléatoirement, dans l'expérience faite dans ce TP, elles varient de 0 à 8. En ce qui concerne l'espace mémoire inutilisée, il n' y a pas de gaspillage de mémoire, sauf lorsque le sauf lorsque le nombre d'éléments ajoutés est inférieur à la capacité du tableau.
- **Quand on fait à la fois un ajout et une extraction des clés dans le tas** :
  - Si p<=0.5, le coût amorti est élevé en début et bas pour des opérations ultérieures (soit moins de 50 dans le cadre de ces expériences). Il n'y a pas assez de permutations, elles varient de 0 à 3 pour p<0.4 et de 0 à 9 pour p=0.5. En ce qui concerne l'espace mémoire inutilisée, il reste élevé à environ la capacité du tableau.
  - Si p>0.5, le coût amorti est élevé en début et bas (au dessus de 50) pour les opérations ultérieures, moins bas que si p>=0.5. Il y a de plus en plus de permutations (entre 0 et 20). En ce qui concerne l'espace mémoire inutilisée, il baisse d'environ 800000 pour p=0.6, d'environ 600000 pour p=0.7, d'environ 400000 pour p=0.8 et d'environ 200000 pour p=0.9.

3)**Expériences sur l'efficacité en temps et en mémoire d'un tas binaire pour un tas représenté par une table dynamique**

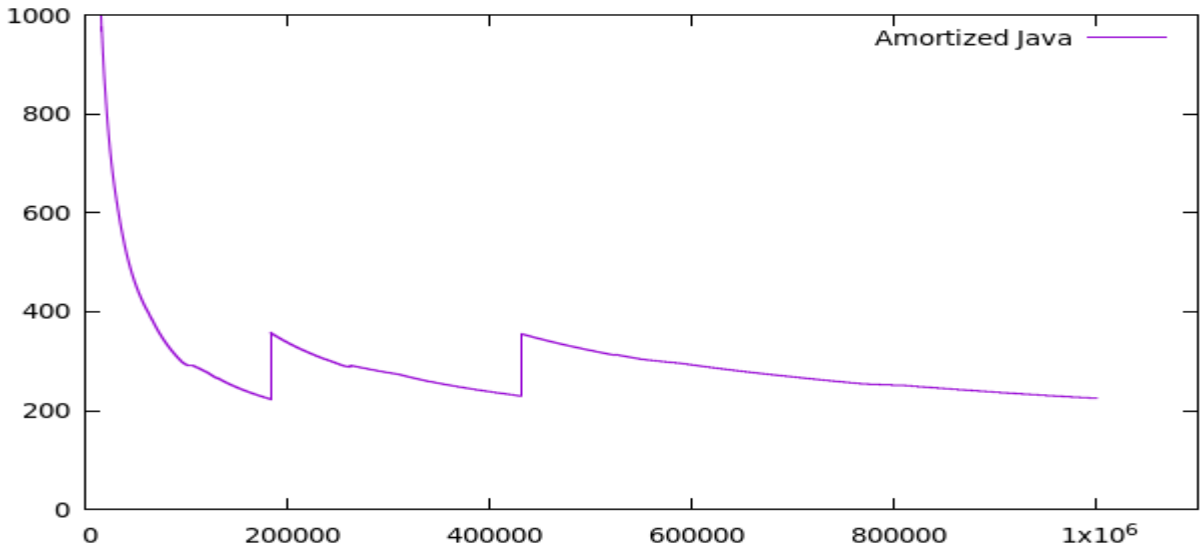
Cas 1 : On ajoute les clés dans l'ordre croissant

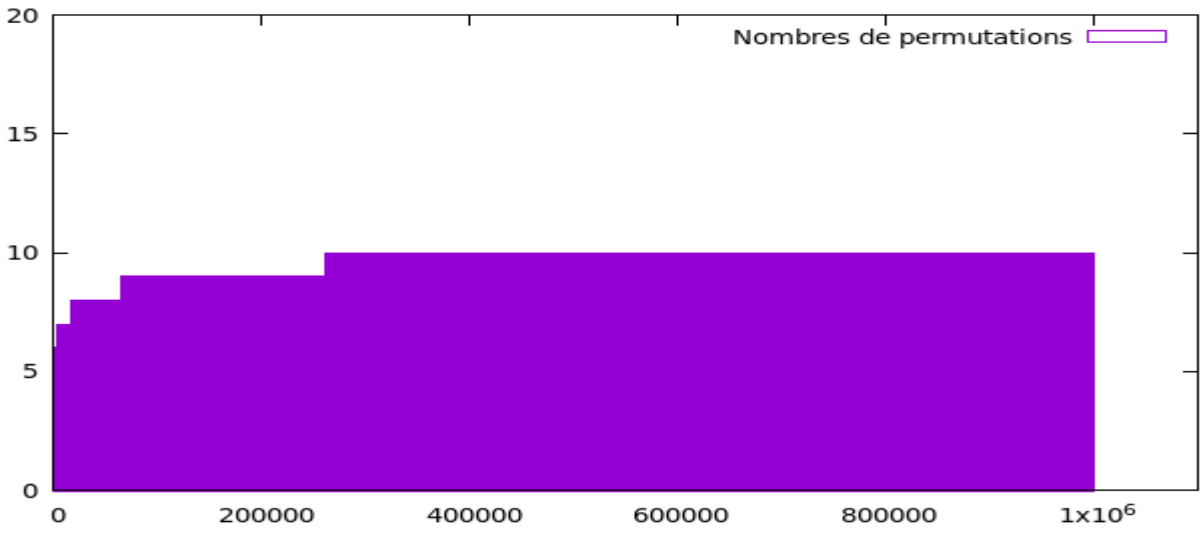


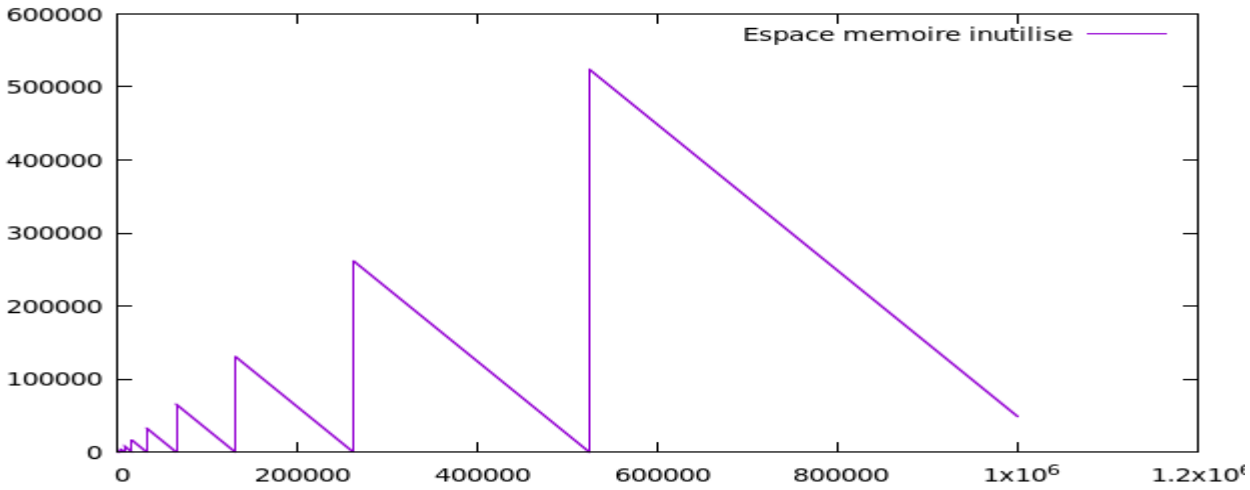




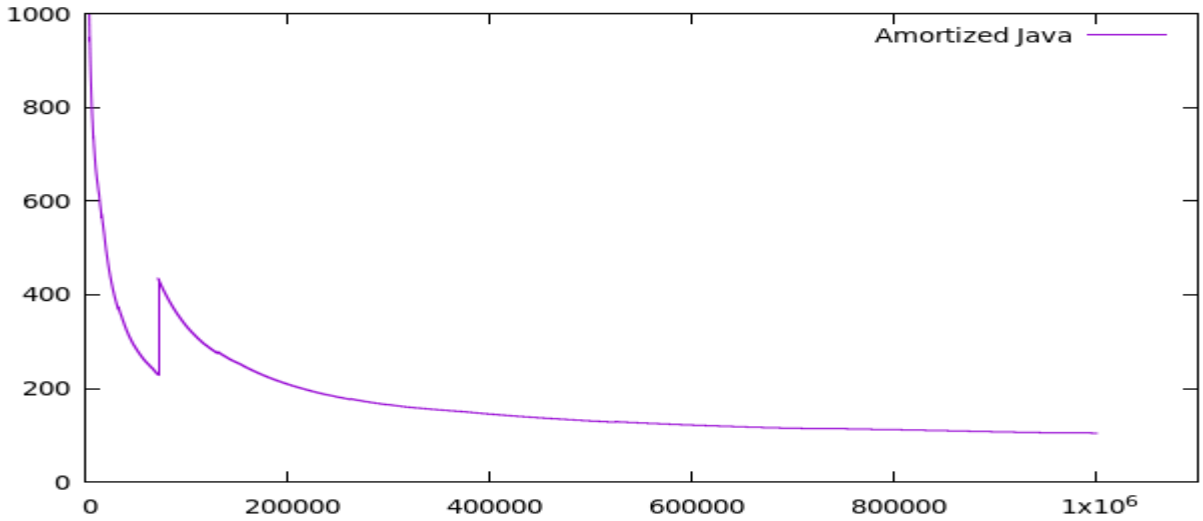
Cas 2 : On ajoute les clés dans l'ordre décroissant

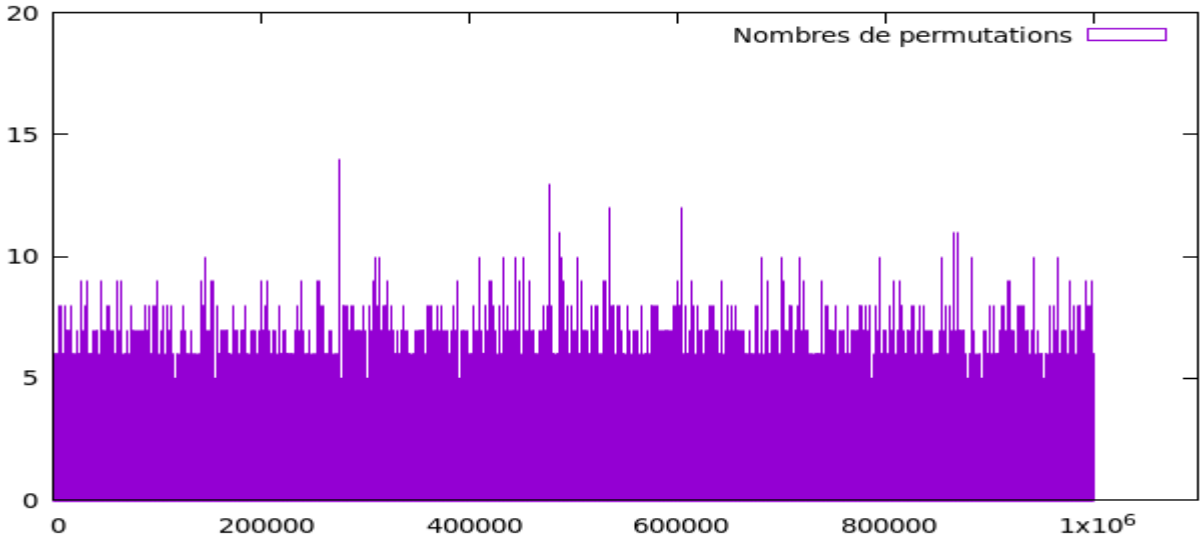


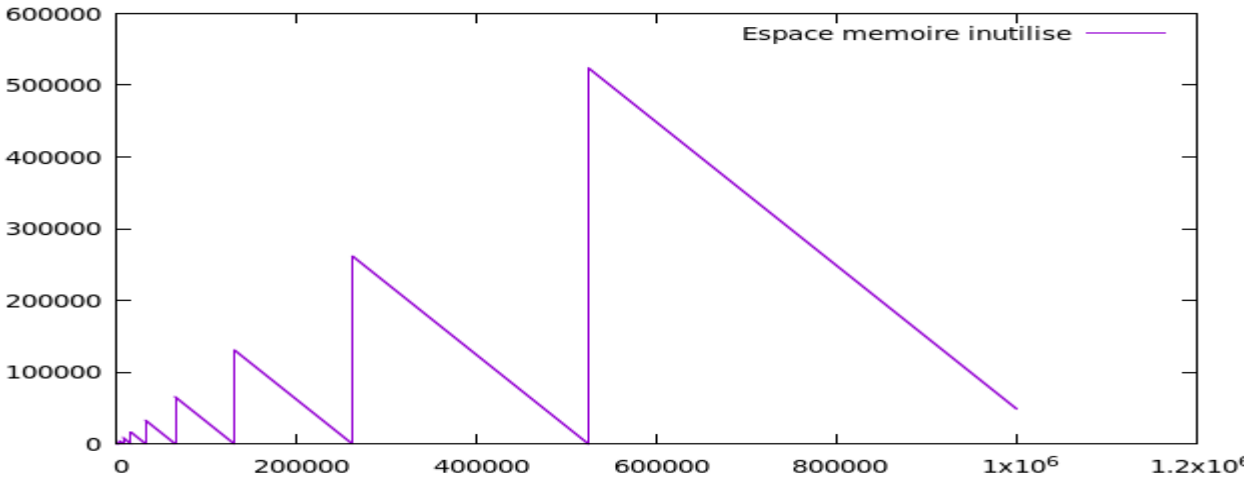




Cas 3 : On ajoute les clés aléatoires

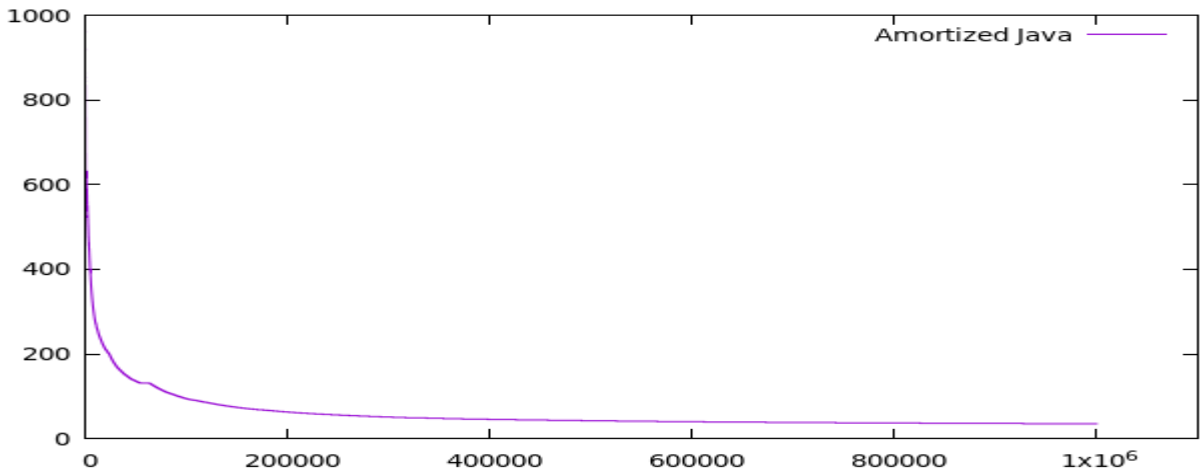


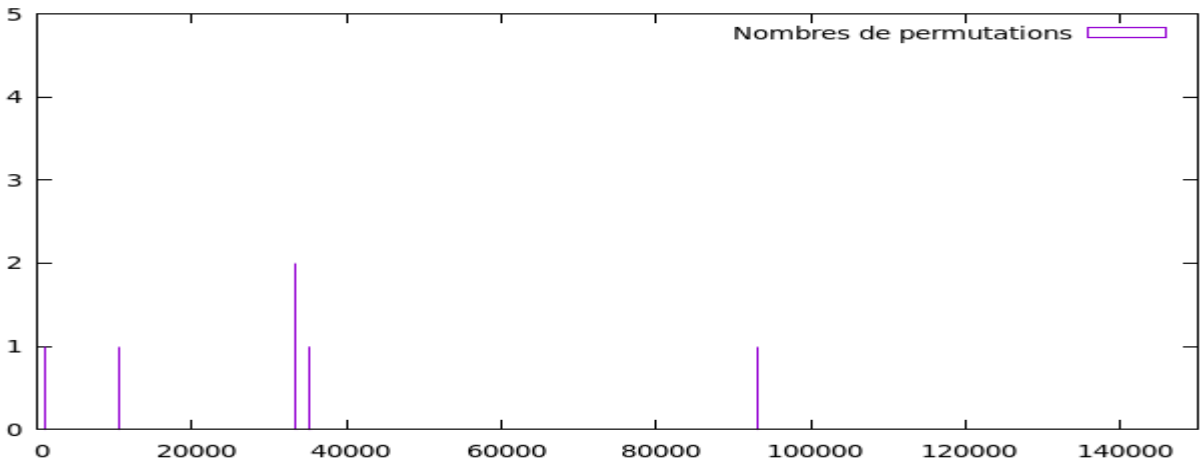




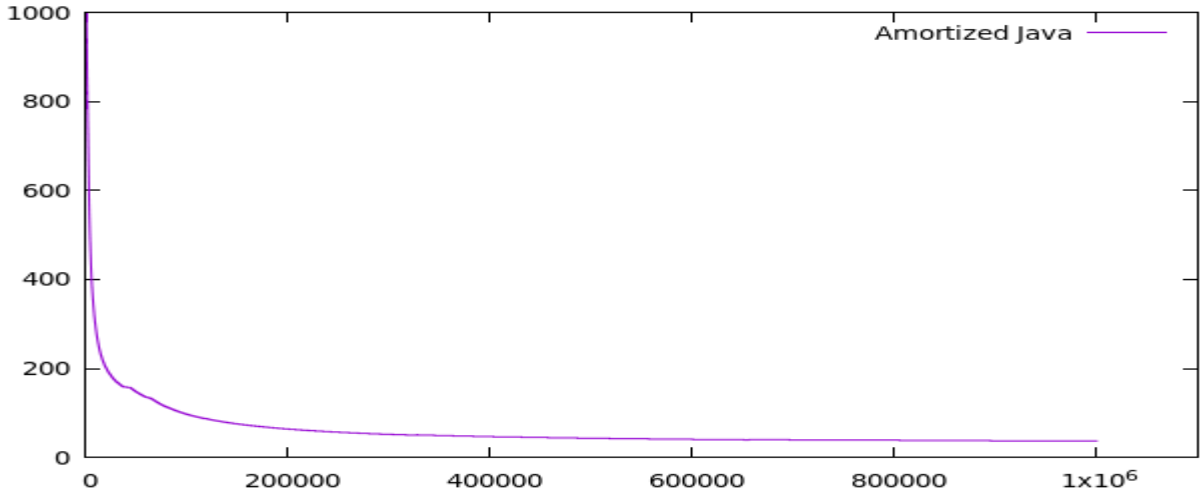
Cas 4 : On ajoute et on extrait les éléments

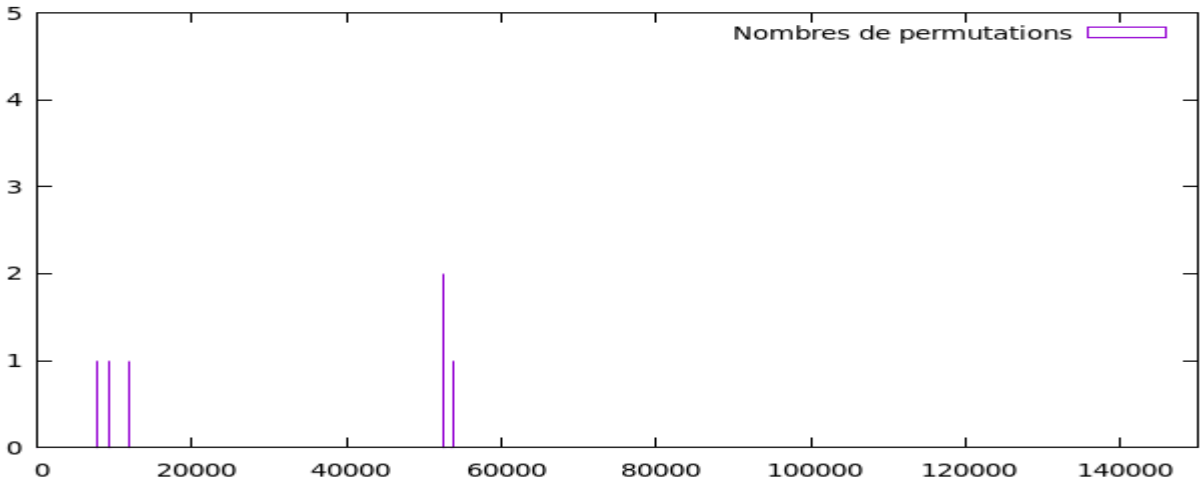
p=0.1



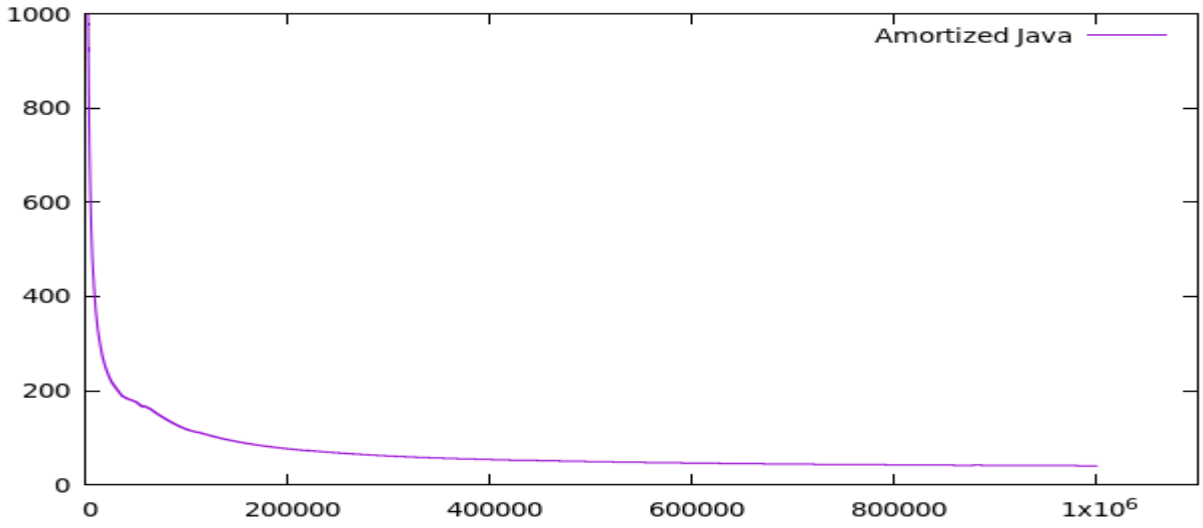


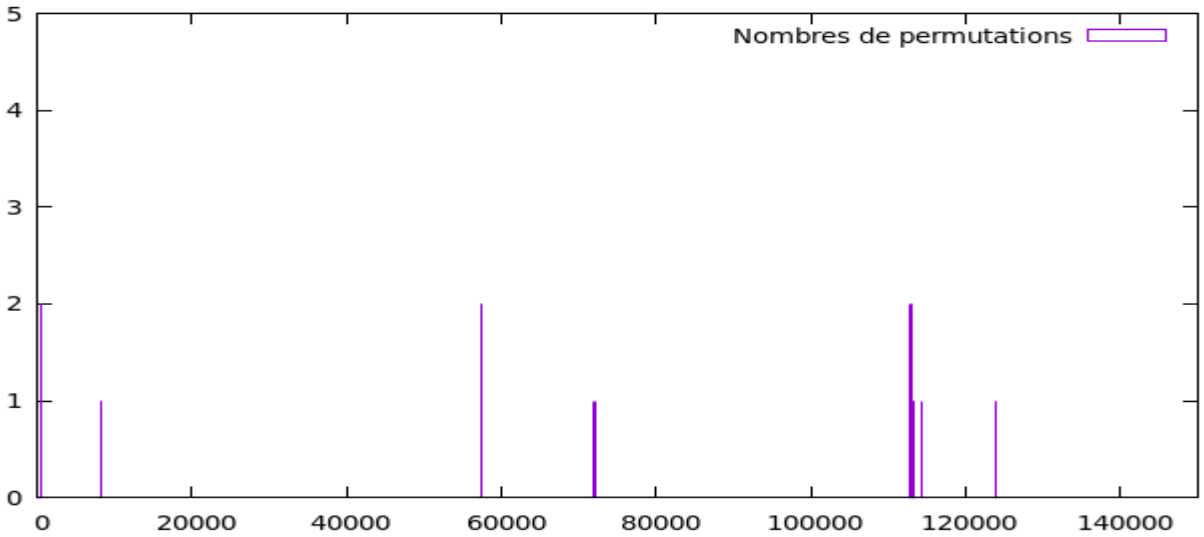
p=0.2

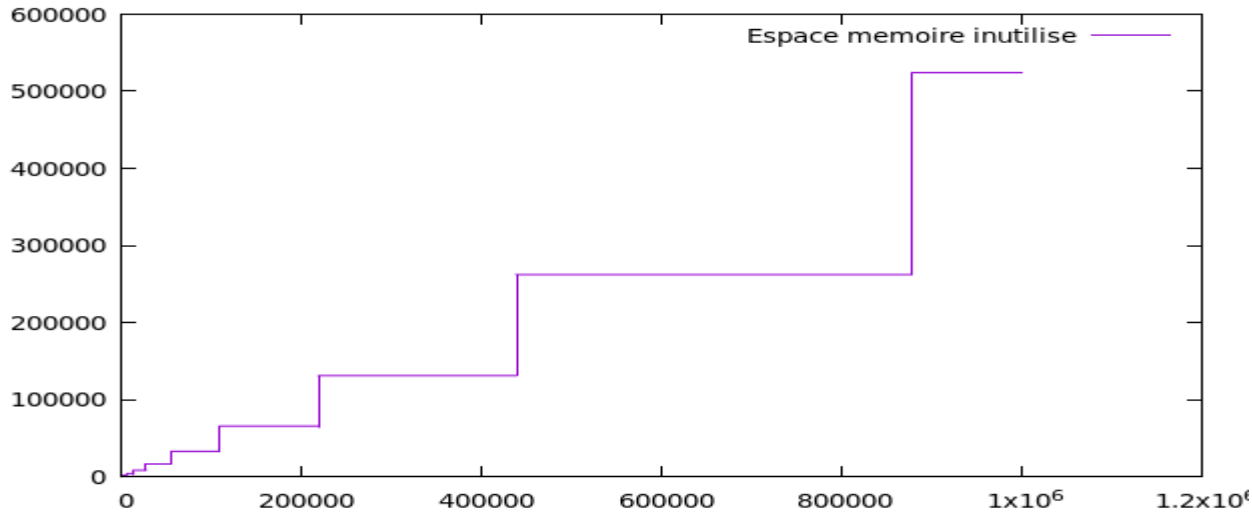

Amortized Java
1000
800
600
400
200
0
0
200000
400000
600000
800000
1x10^6


Nombres de permutations
5
4
3
2
1
0
0
20000
40000
60000
80000
100000
120000
140000

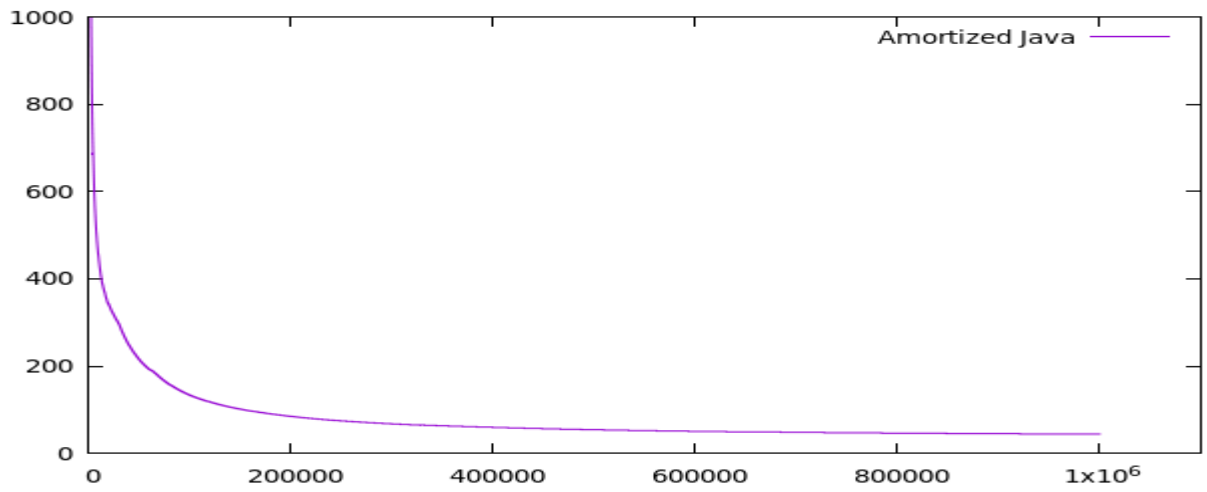
p=0.3

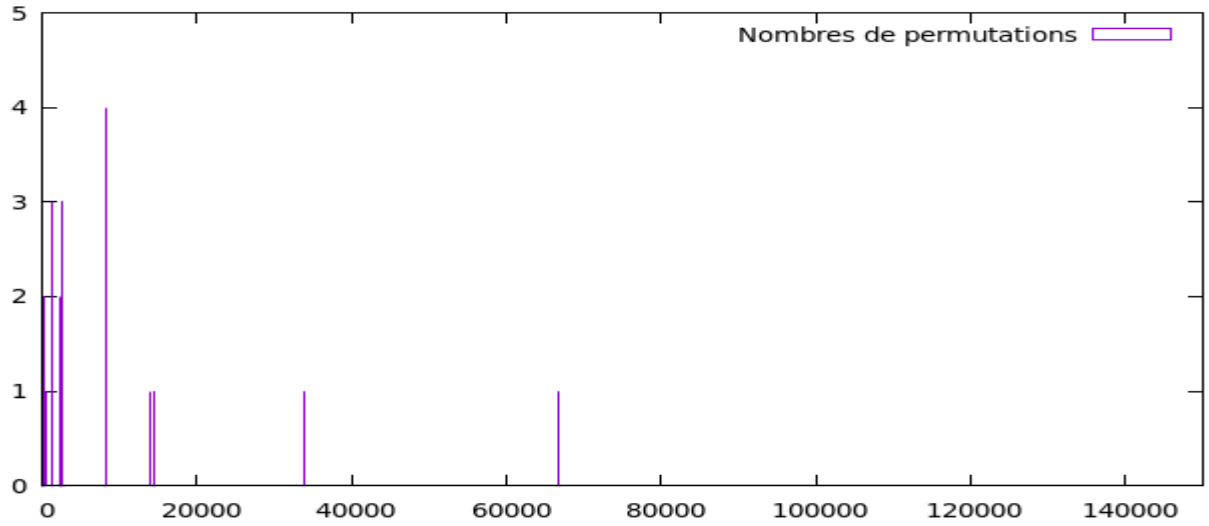

Amortized Java
1000
800
600
400
200
0
0
200000
400000
600000
800000
1x10^6

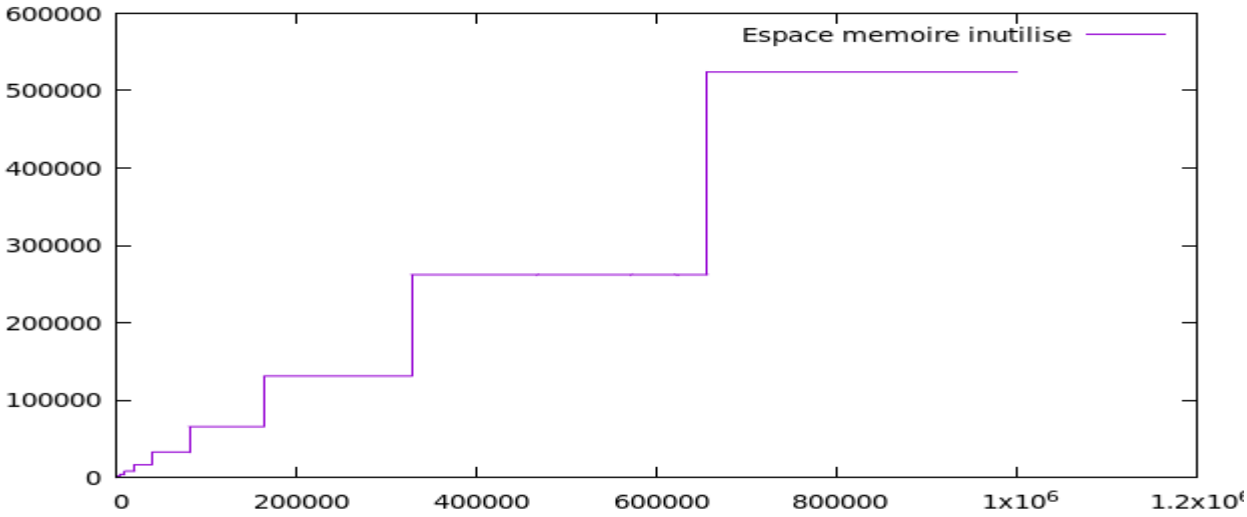

Nombres de permutations
5
4
3
2
1
0
0
20000
40000
60000
80000
100000
120000
140000


Espace memoire inutilise
600000
500000
400000
300000
200000
100000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

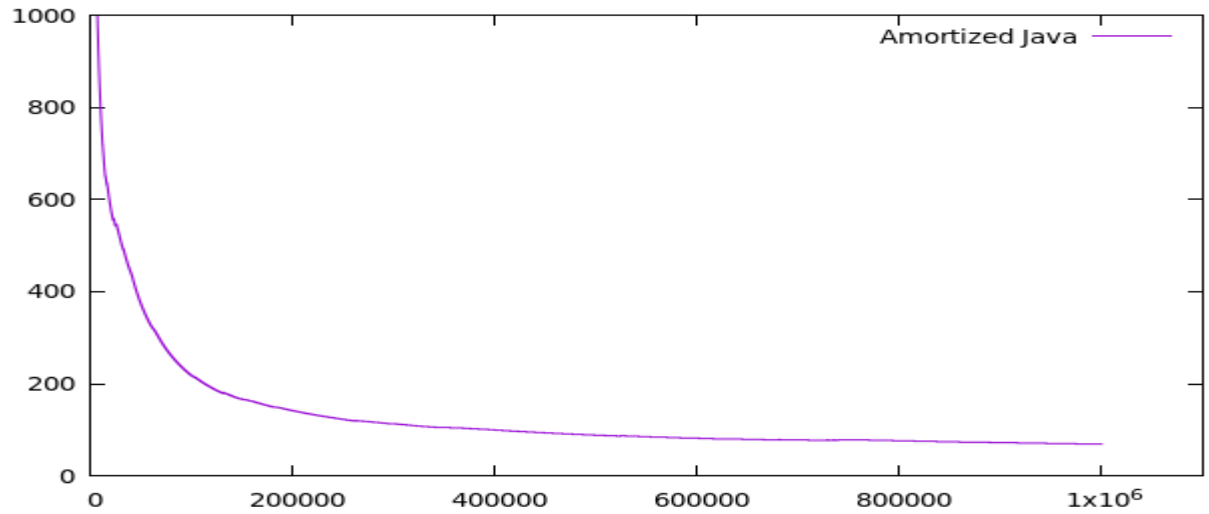
p=0.4

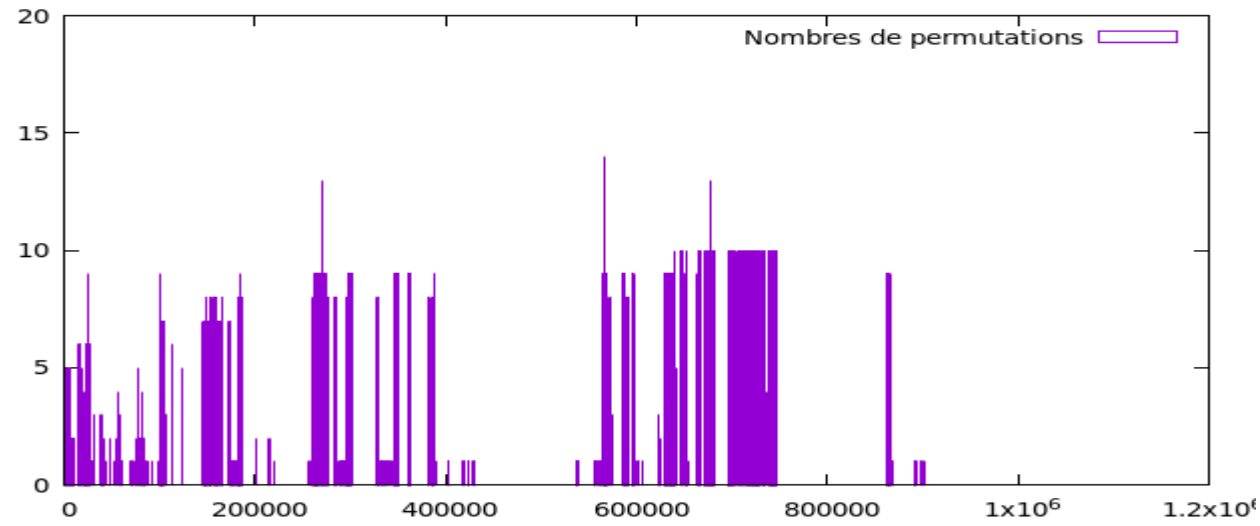

Amortized Java
1000
800
600
400
200
0
0
200000
400000
600000
800000
1x10^6

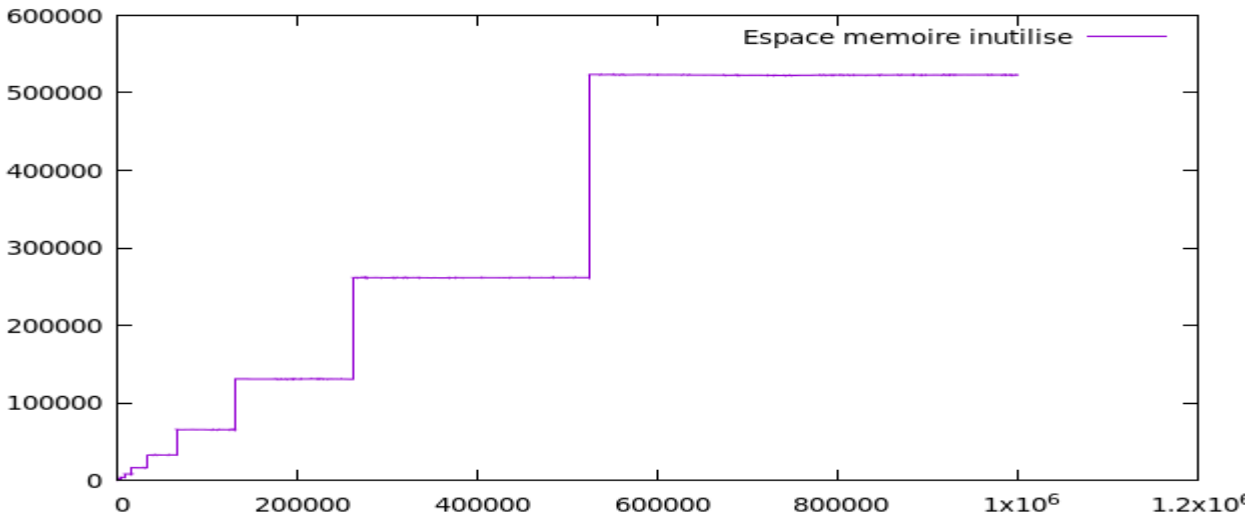

Nombres de permutations
5
4
3
2
1
0
0
20000
40000
60000
80000
100000
120000
140000


Espace memoire inutilise
600000
500000
400000
300000
200000
100000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

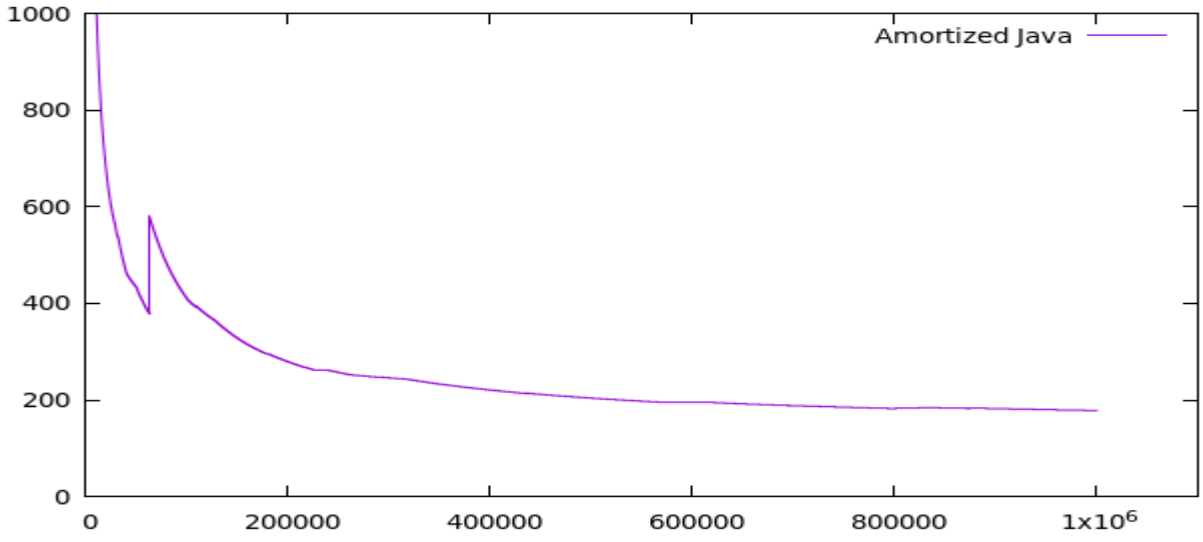
p=0.5

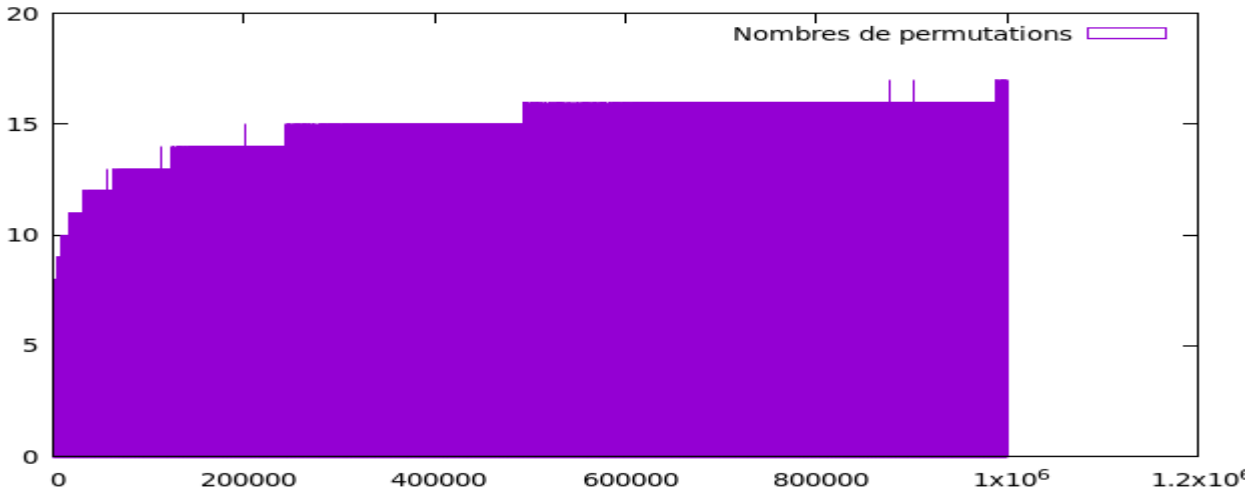

Amortized Java
1000
800
600
400
200
0
0
200000
400000
600000
800000
1x10^6

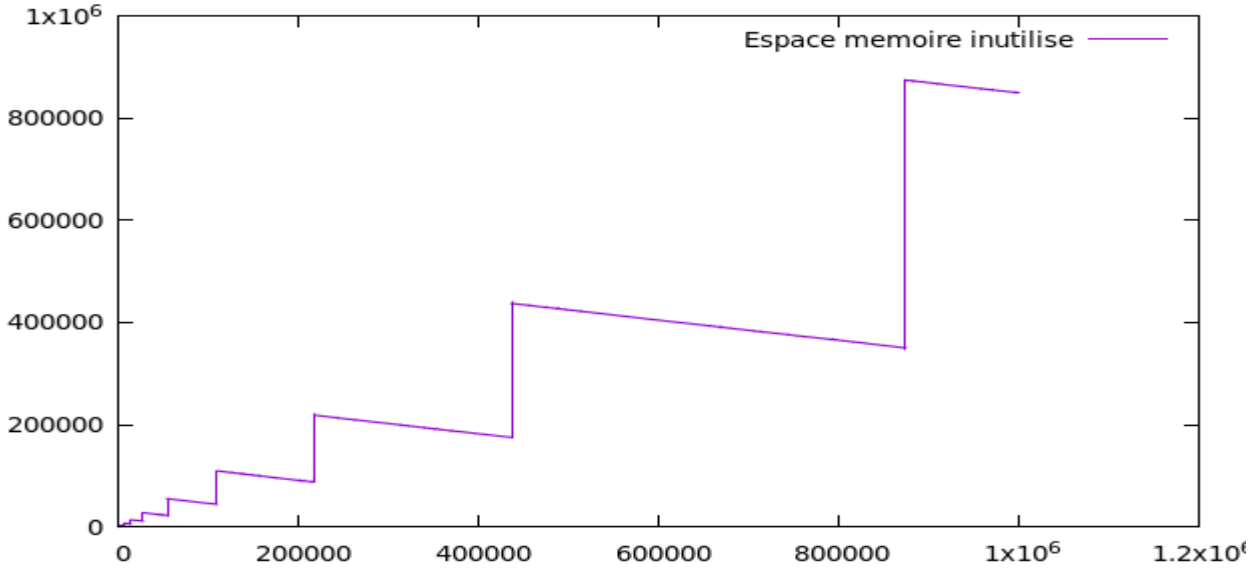

Nombres de permutations
20
15
10
5
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6


Espace memoire inutilise
600000
500000
400000
300000
200000
100000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

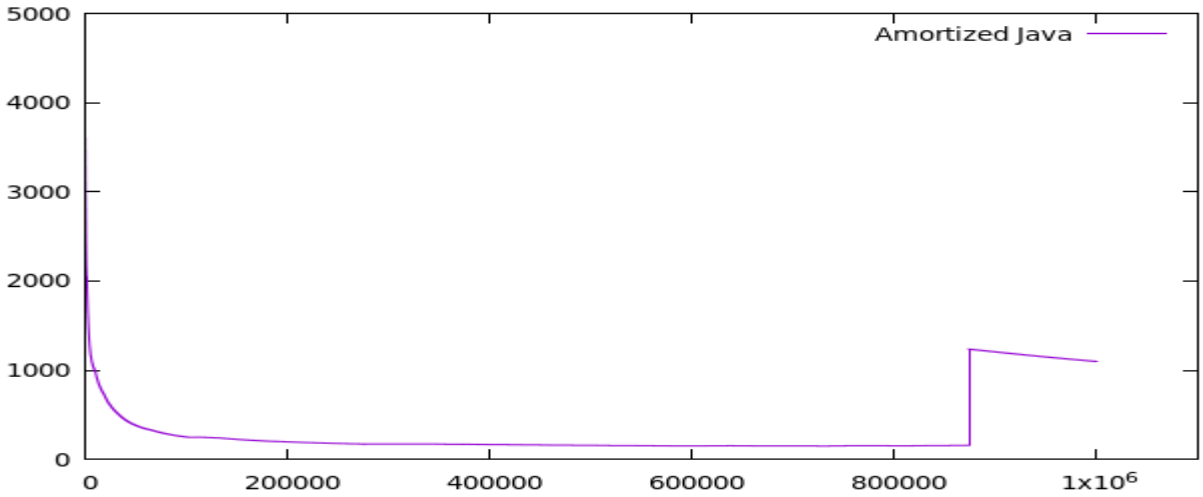
p=0.6

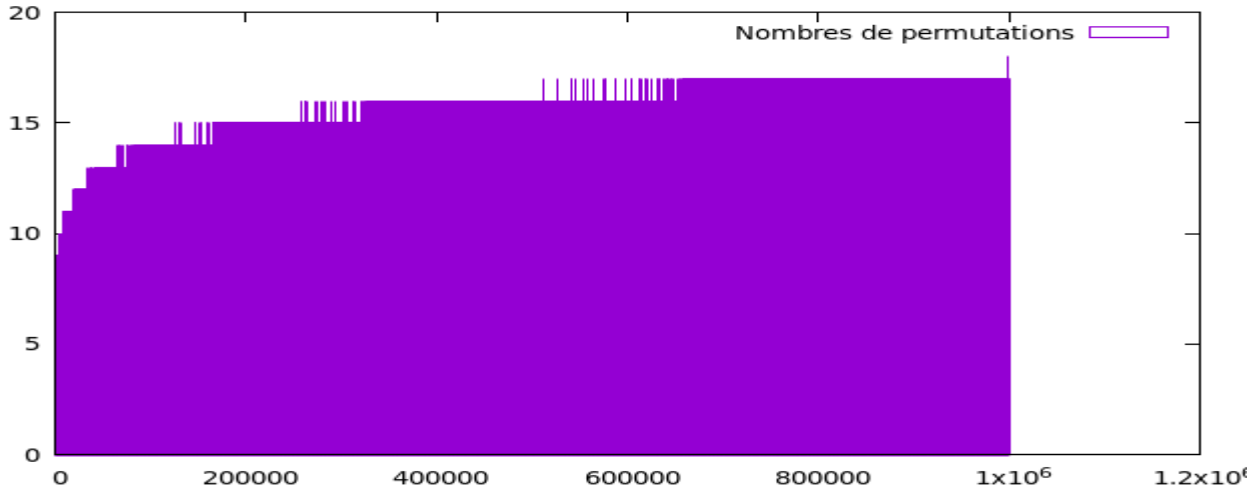

Amortized Java
1000
800
600
400
200
0
0
200000
400000
600000
800000
1x10^6

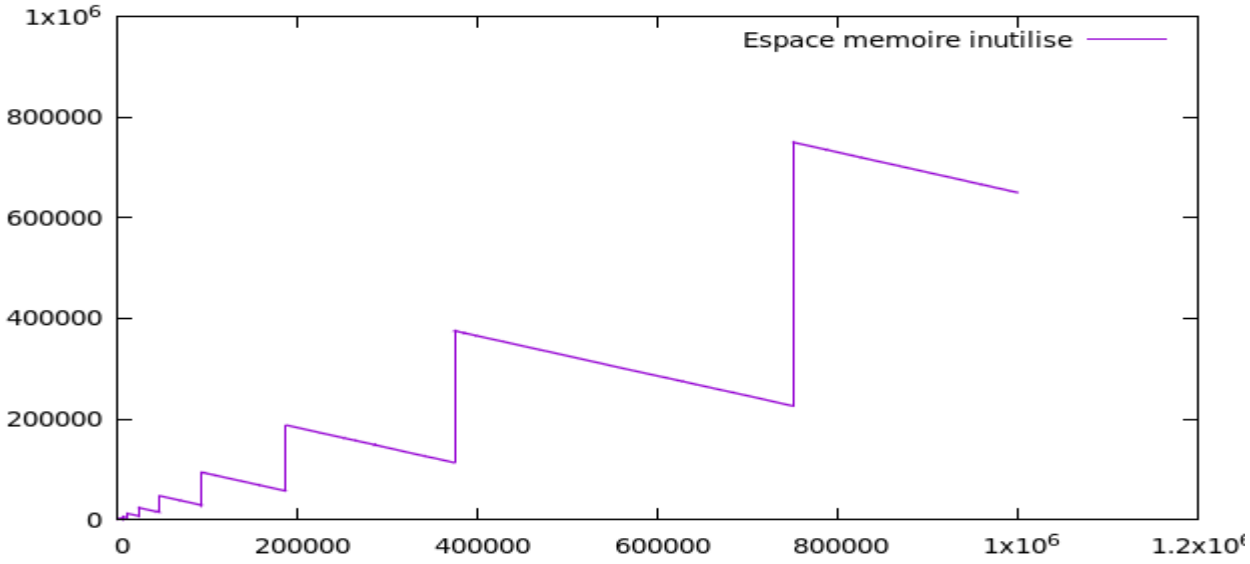

Nombres de permutations
20
15
10
5
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6


Espace memoire inutilise
1x10^6
800000
600000
400000
200000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

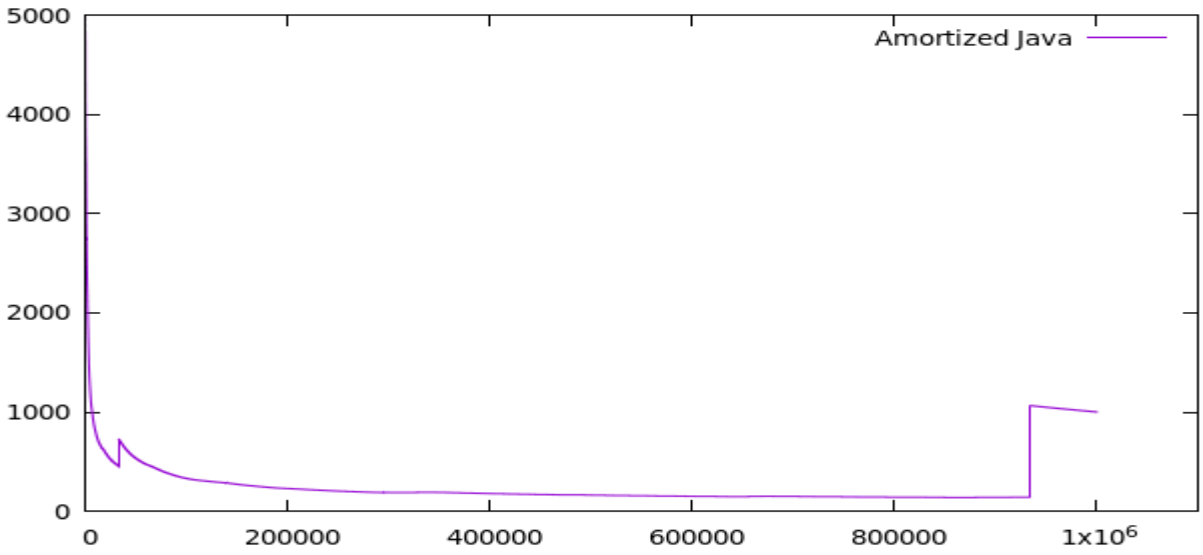
p=0.7

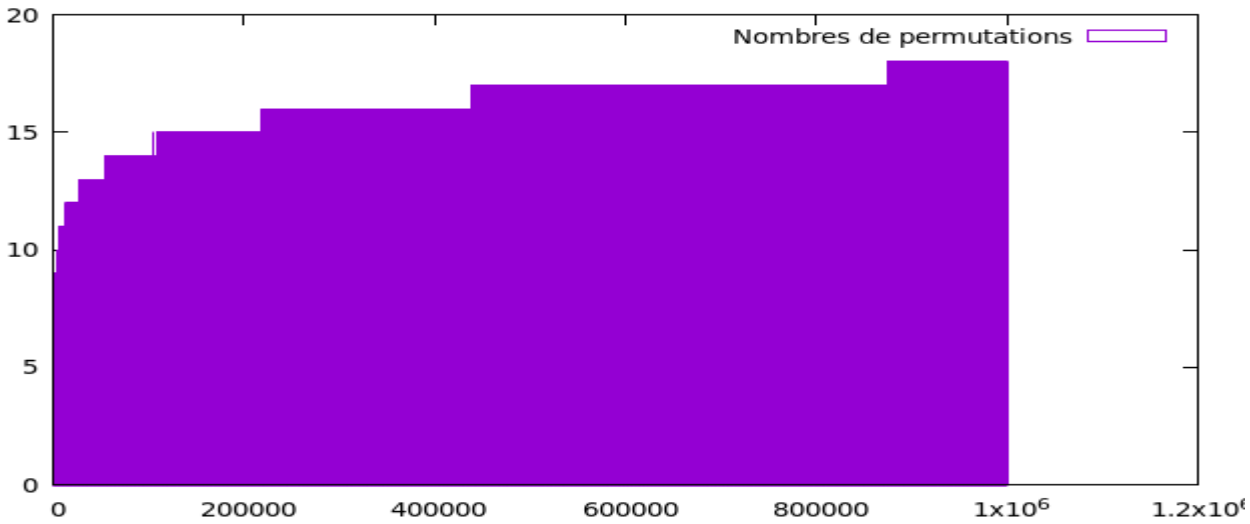

Amortized Java
5000
4000
3000
2000
1000
0
0
200000
400000
600000
800000
1x10^6

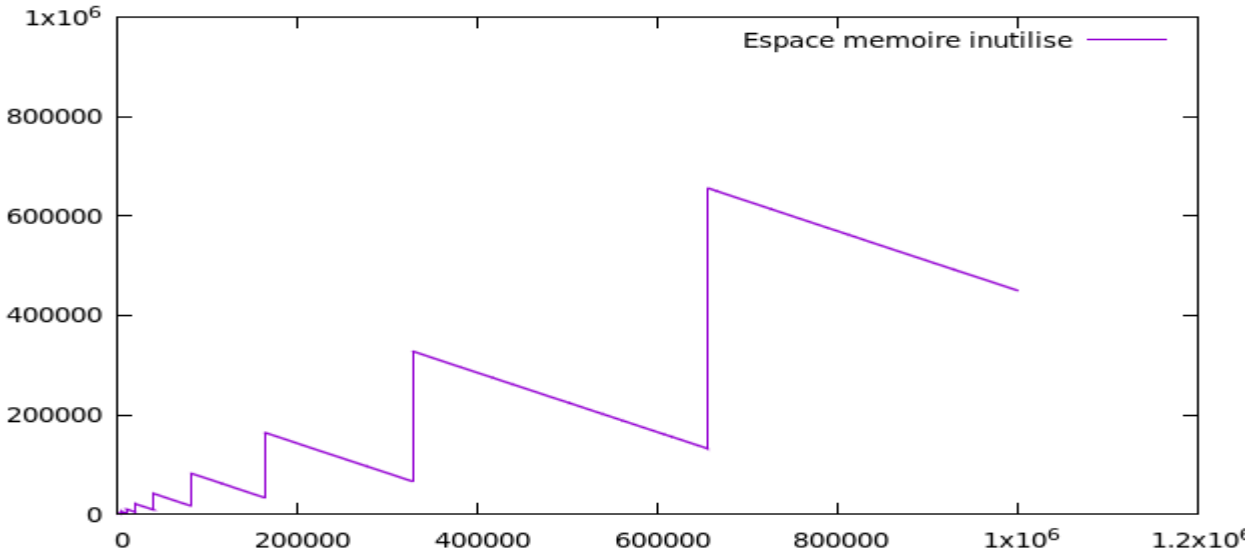

Nombres de permutations
20
15
10
5
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6


Espace memoire inutilise
1x10^6
800000
600000
400000
200000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

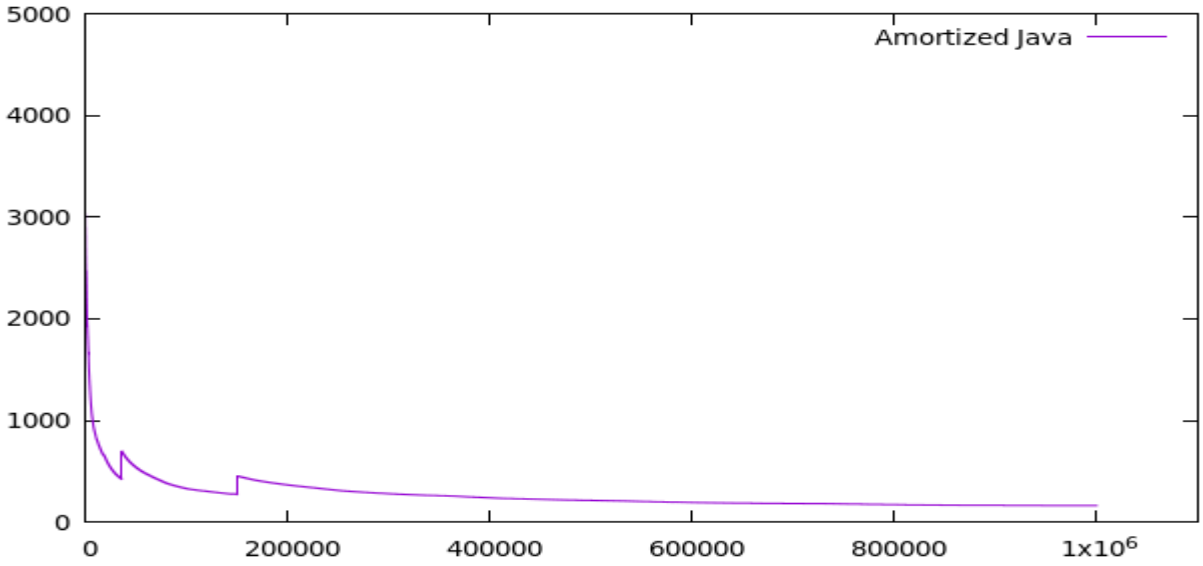
p=0.8

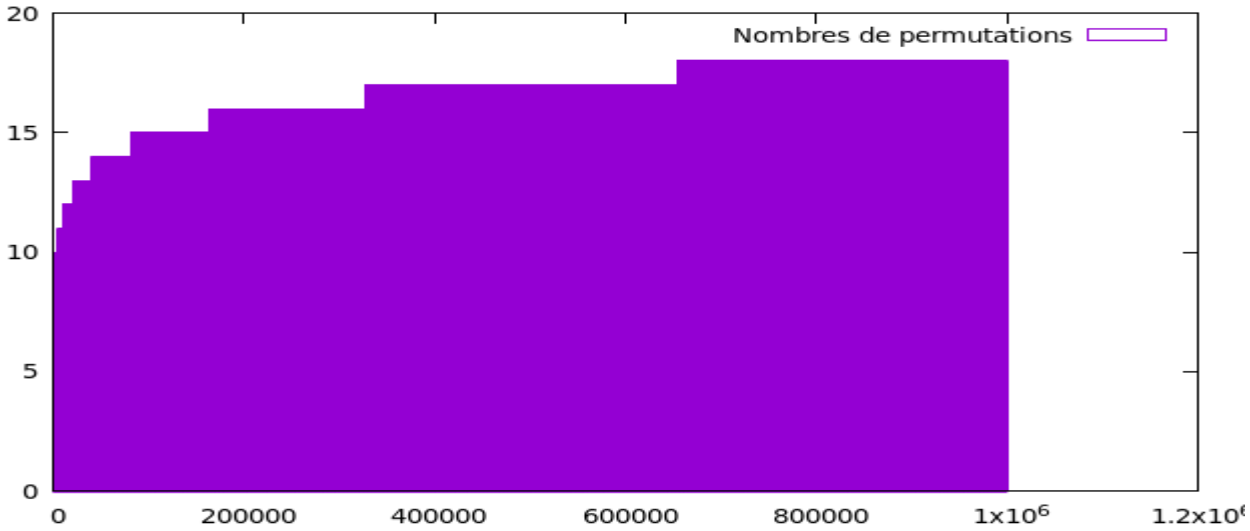

Amortized Java
5000
4000
3000
2000
1000
0
0
200000
400000
600000
800000
1x10^6

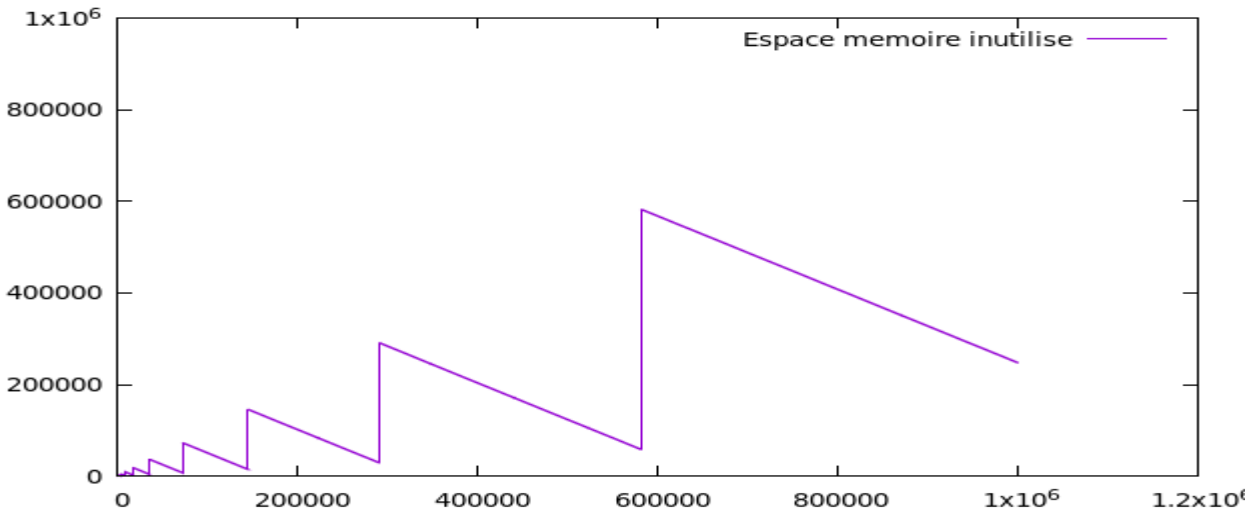

Nombres de permutations
20
15
10
5
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6


Espace memoire inutilise
1x10^6
800000
600000
400000
200000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

p=0.9


Amortized Java
5000
4000
3000
2000
1000
0
0
200000
400000
600000
800000
1x10^6


Nombres de permutations
20
15
10
5
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6


Espace memoire inutilise
1x10^6
800000
600000
400000
200000
0
0
200000
400000
600000
800000
1x10^6
1.2x10^6

Analyse

Dans le cas où le tas est représenté dans une table dynamique,

- **Quand on ajoute les clés par ordre croissant**, le coût amorti est élevé en début et diminue pour les opérations ultérieures, mais pas assez par rapport au cas où c'est un tableau de taille fixe. Il n' y a pas du tout de permutations entre les clés. Il reste des cases mémoires inutilisées (soit environ 50000)
- ➔ **Quand on ajoute les clés par ordre décroissant**, le coût amorti est élevé en début et diminue pour les opérations ultérieures, mais pas assez par rapport au cas où c'est un tableau de taille fixe. Il y a de plus en plus de permutations entre les clés (elles varient de 1 à 9). L'espace mémoire inutilisée augmente davantage (environ 200000).

Quand on ajoute les clés de façon aléatoire, le coût amorti est élevé en début et diminue pour les opérations ultérieures, mais pas assez par rapport au cas où c'est un tableau de taille fixe. Il y a de plus en plus de permutations entre les clés (elles varient de 1 à 7). L'espace mémoire inutilisée est aussi faible.

Quand on fait à la fois l'ajout et l'extraction des clés dans le tas,

- ◆ Si $p<0.5$, le coût amorti est élevé au début et faible pour les opérations ultérieures. Les permutations entre les clés sont très rares mais quand elles arrivent elles sont faibles (On a entre 1 et 2 pour $p=0.1$, $p= 0.2$, $p=0.3$; 1 et 4 pour $p=0.4$). En ce qui concerne l'espace mémoire inutilisée, il augmente au fur et à mesure que p est grand. Pour $p=0.1$, il est d'environ 150000, pour $p=0.2$, il est d'environ 250000, pour $p=0.3$ et $p=0.4$, il est d'environ 500000.
- ◆ Si $p>=0.5$, le coût amorti est élevé au début et faible pour les opérations ultérieures, mais moins faible si $p<0.5$. En gros, il est plus élevé dans ce cas-ci exception faite dans le cas où $p=0.9$. Il arrive qu'il augmente pour les opérations ultérieures (le cas de $p=0.7$). Il y a davantage des permutations entre les clés dans le tableau (soit de 1 à 14 pour $p=0.5$, de 1 à 17 pour $p=0.6$, $p=0.7$, $p=0.8$, $p=0.9$). L'espace mémoire inutilisée continue d'augmenter jusqu'à $p=0.6$ (soit 500000 pour $p=0.5$, 800000 pour $p=0.6$) puis diminue à partir de $p=0.7$ jusqu'à $p=0.9$ (soit 600000 pour $p=0.7$, 400000 pour $p=0.8$, 200000 pour $p=0.9$).

<u>Complexité amortie des opérations ajout/extraction</u>

La complexité amortie des opérations ajout/ extraction a changé dans le cas où $p>0.5$. En effet, elle est élevée en début et pour les opérations ultérieures. Cela est dû au fait que l'espace mémoire inutilisée est plus grand.

Synthèse

A travers ce TP, nous avons fait une analyse des coûts des opérations dans une structure de tas binaire. Le tas binaire pouvant être représenté par un tableau. Nos expériences ont porté sur 2 types de tableau : le tableau de taille fixe et le tableau de taille dynamique.

Quand un tas est représenté sous forme de tableau de taille fixe

Dans le cas où les clés sont ajoutés par ordre croissant, le coût amorti est au plus bas pour les opérations ultérieures, il n' y a pas du tout de permutations car les clés ajoutées par ordre

croissant vérifient bien les propriétés des tas min. Aussi, il n'y a pas de gaspillage de mémoire sauf lorsque le nombre d'éléments ajoutés est plus petit que la capacité du tas.
Dans le cas où les clés sont ajoutés par ordre décroissant, le coût amorti est moins bas que lorsque les clés sont croissantes. On constate une légère augmentation des permutations. En ce qui concerne l'espace mémoire inutilisée, il n' y a pas de gaspillage de mémoire.
Dans le cas où les clés ajoutées sont tirées aléatoirement, le coût amorti est moins bas que lorsque les clés sont décroissantes. Il n' y a pas gaspillage de mémoire, aussi il est difficile de prévenir le nombre de permutations possibles, néanmoins on enregistre une augmentation.
Dans le cas où on a une suite de n opérations d'insertions et d'extractions dans le tas,
Si $p<0.5$, le coût amorti diminue au plus bas, il n'y a pas assez de permutations, ou presque pas, selon les cas. Quant à l'espace mémoire inutilisé, il y a un grand gaspillage de mémoire, environ la capacité du tableau.
Si $p>0.5$, le coût amorti diminue moins que dans le cas précédent, dire qu'il est plus élevé que dans le cas où $p<0.5$. Il y a de plus en plus de permutations mais de moins en moins de gaspillage de mémoire.

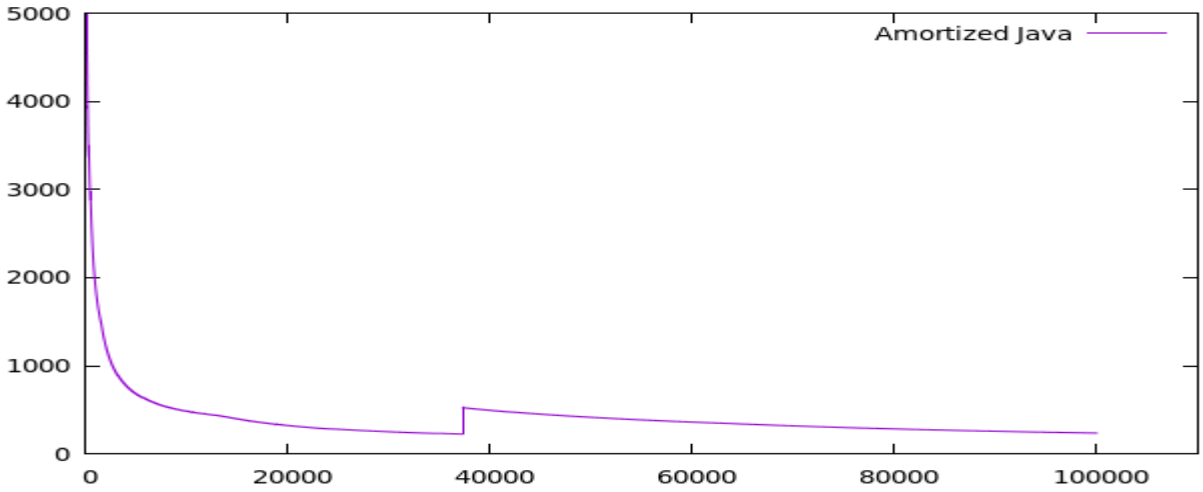
Quand un tas est représenté sous forme de tableau de taille dynamique,
Dans le cas où on ajoute les clés par ordre croissant, le cas où on ajoute les clés par ordre décroissant, le cas où on ajoute les clés de façon aléatoire, le coût amorti ne diminue pas assez. En gros, le coût amorti reste élevé. Il n'y a pas du tout de permutation, par contre il reste des cases mémoires inutilisées. En revanche, en ce qui concerne le premier cas, où on ajoute les clés par ordre croissant, il n' y a du tout de permutation.

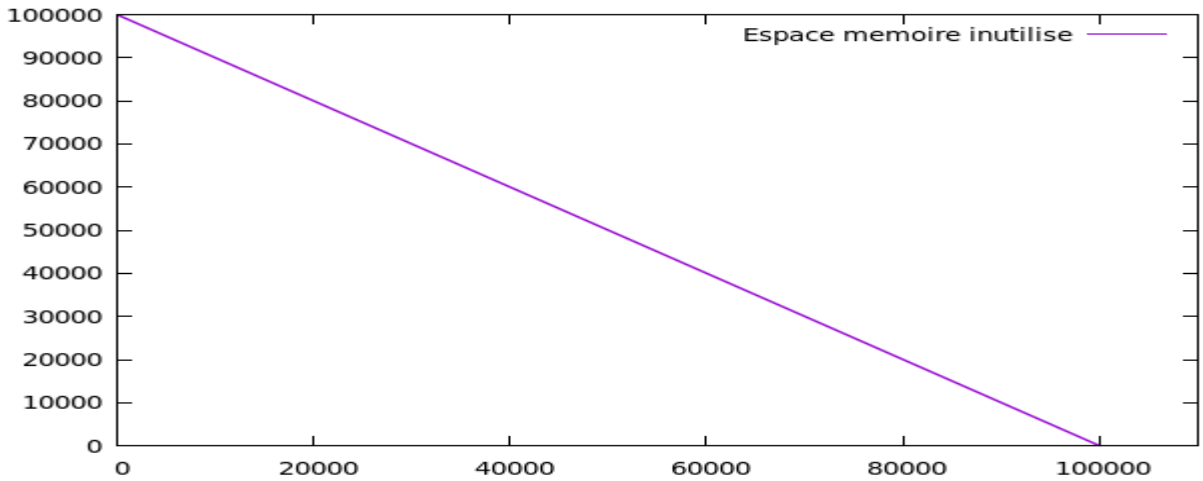
Dans le cas où on a une suite de n opérations d'insertions et d'extractions dans le tas,
Si $p<0.5$, le coût amorti est élevé au début et faible pour des opérations ultérieures, il y a moins de permutations ou presque pas. En ce qui concerne l'espace mémoire inutilisée, on constate moins de gaspillage de mémoire, néanmoins il ne diminue jamais, il augmente au fur et à mesure qu'on a des opérations.
Si $p>=0.5$, le coût amorti amorti est élevé au début et à la fin de certaines opérations, aussi il est faible pour les opérations ultérieures mais moins faible que dans le cas où $p<0.5$. Il y a augmentation du nombre de permutation entre les clés. Quant à l'espace mémoire inutilisée, il est de plus en plus grand. Il augmente quand il reste un certain nombre de cases mémoires.

II - TAS BINOMIAUX

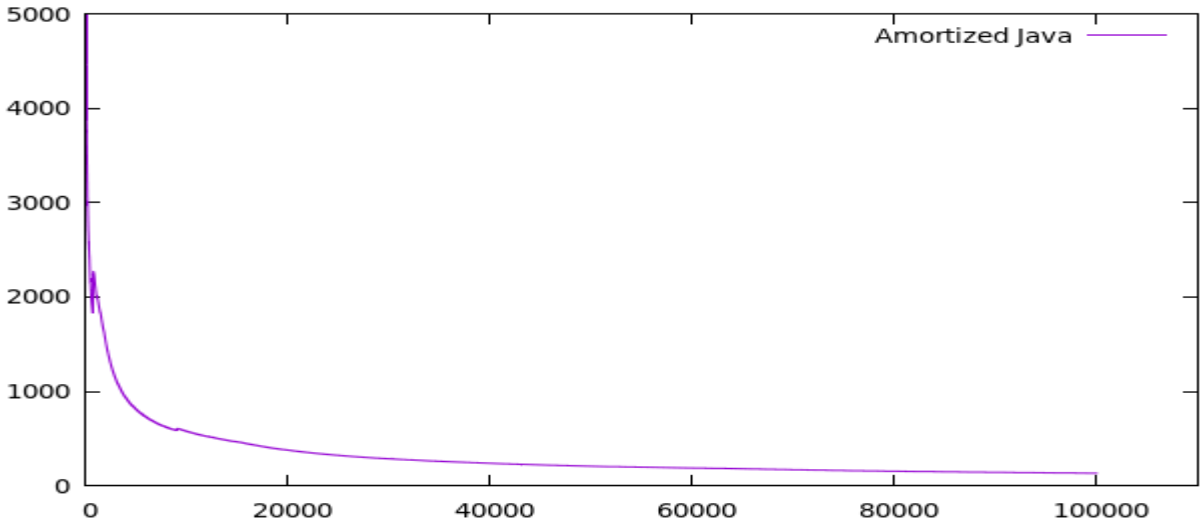
3) **Ajout des clés dans un tas binomial**

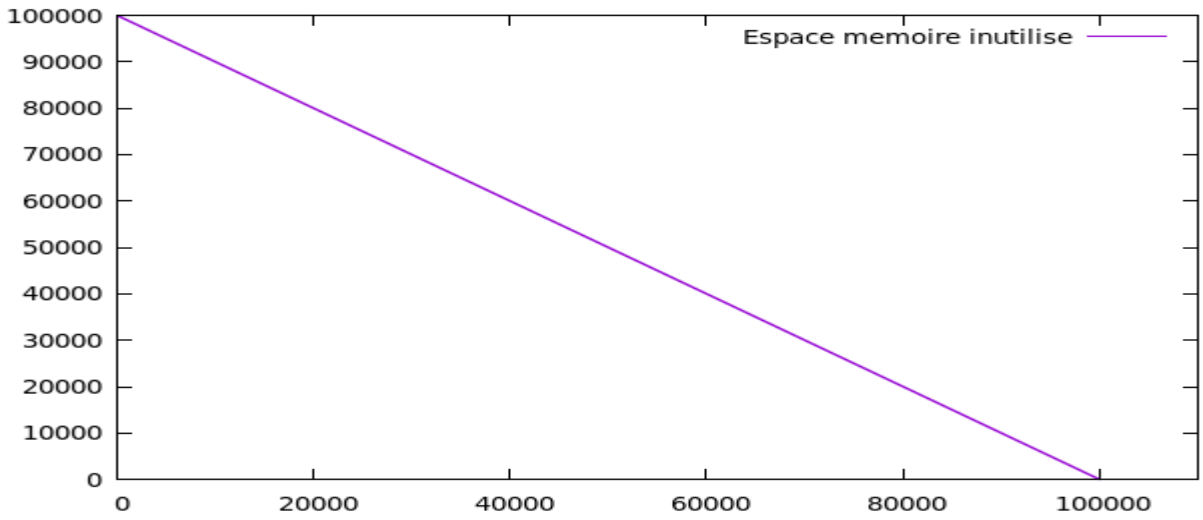
Cas 1 : les clés sont ajoutées dans l'ordre croissant



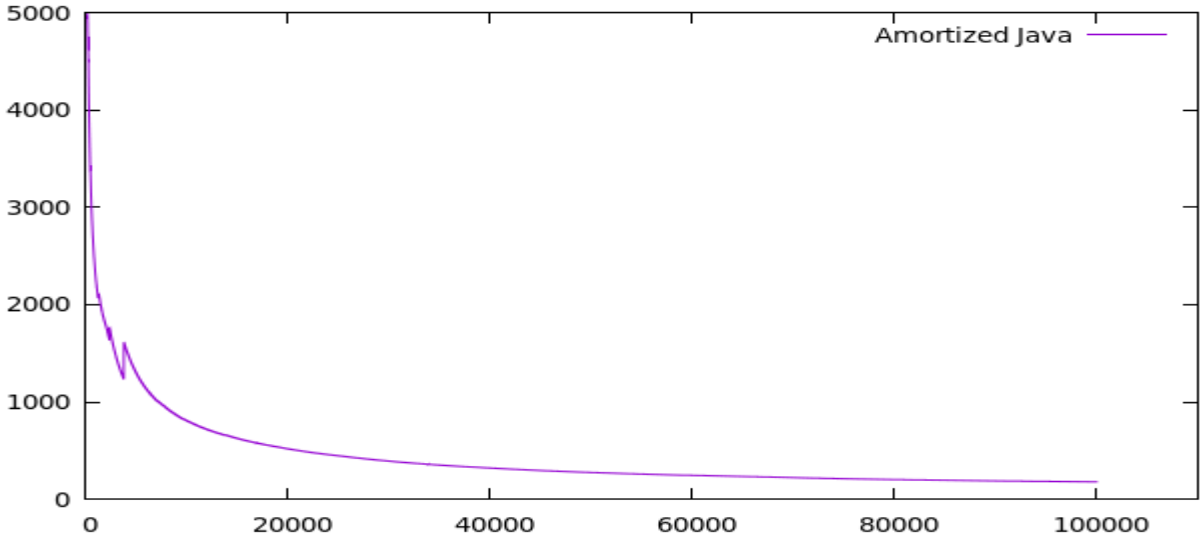


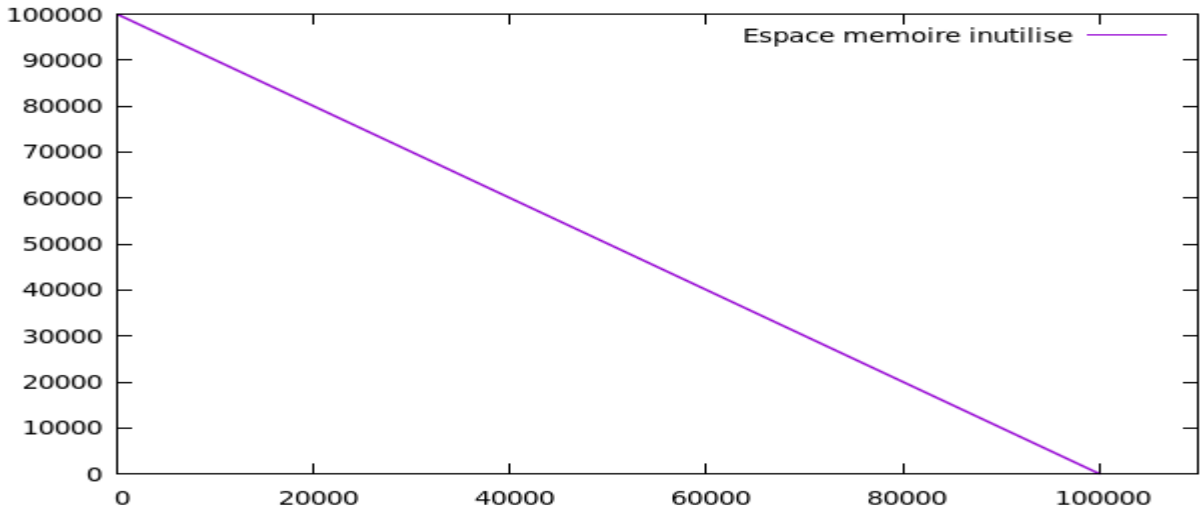
Cas 2 : Les clés sont ajoutées dans l'ordre décroissant





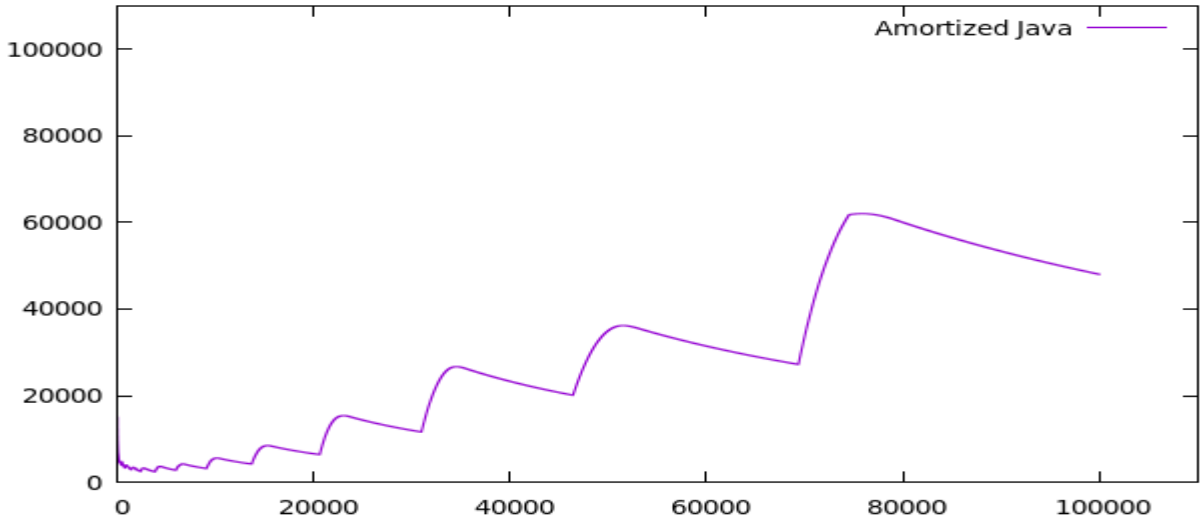
Cas 3 : Les clés sont ajoutées aléatoirement

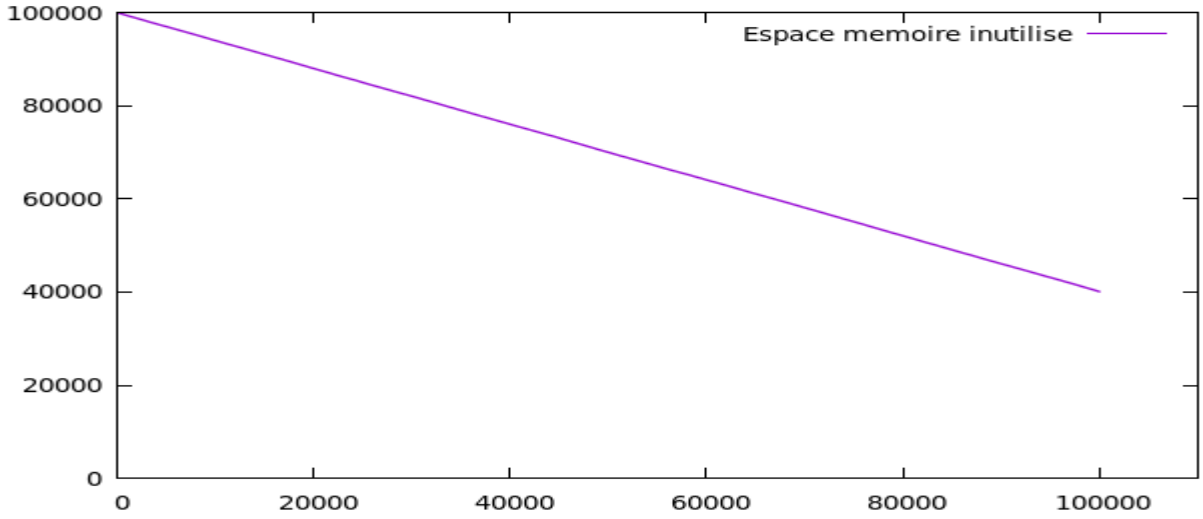




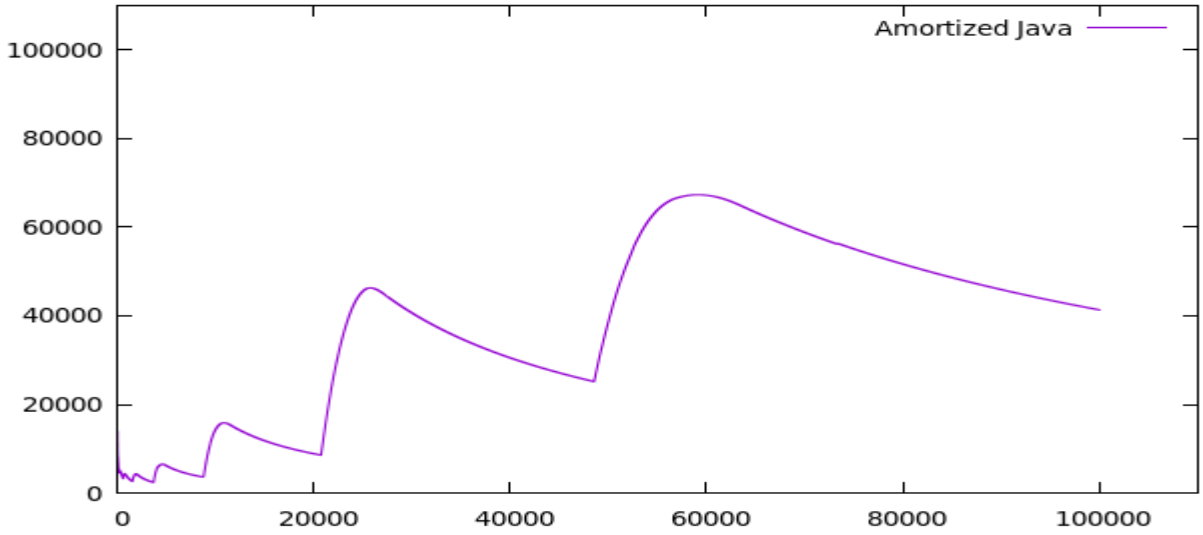
Ajouts et extractions

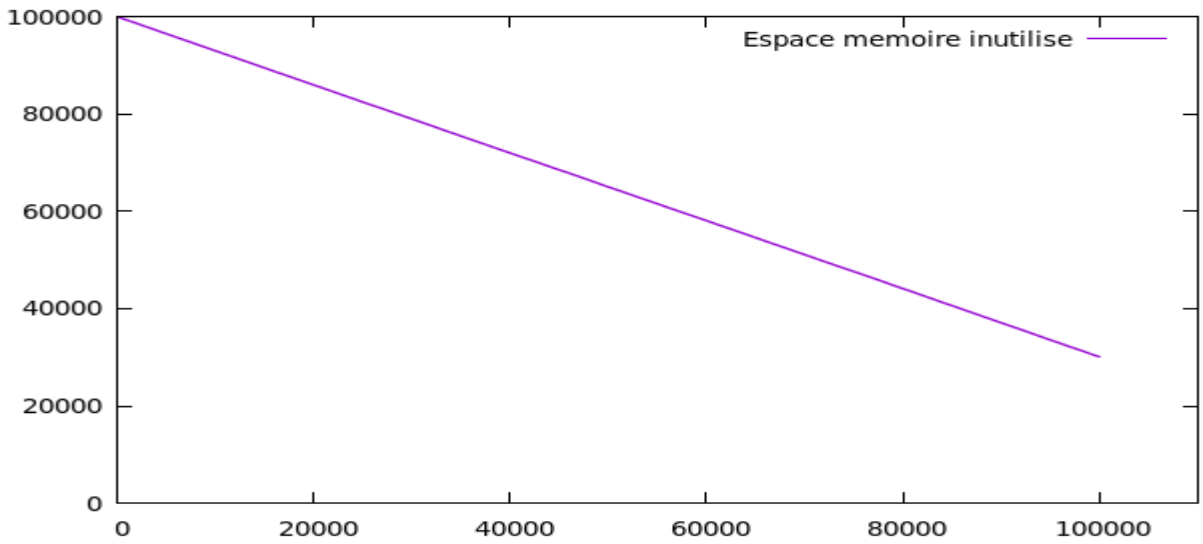
p=0.6



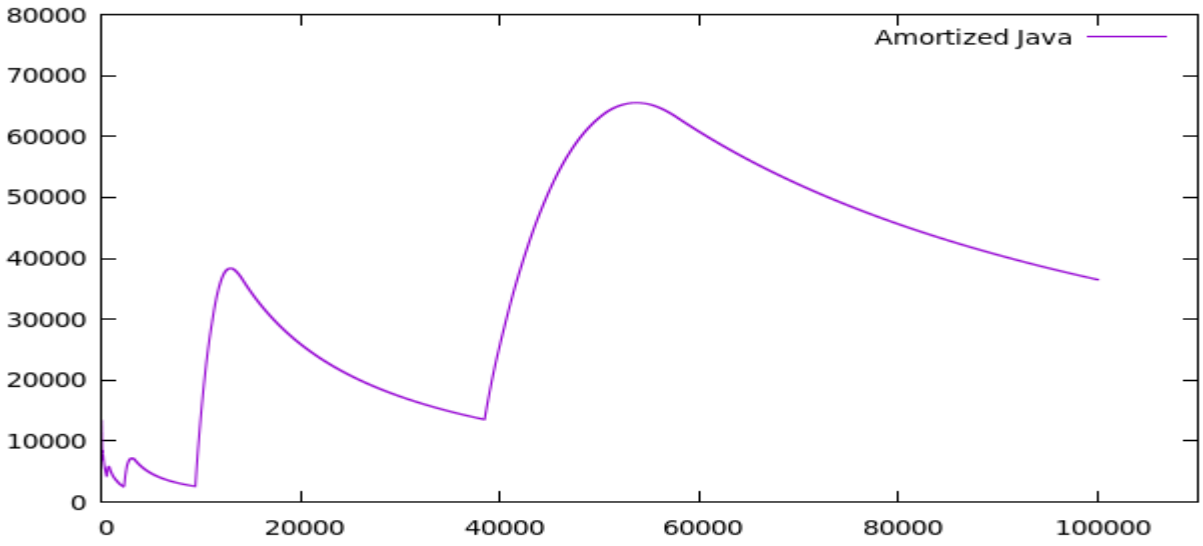


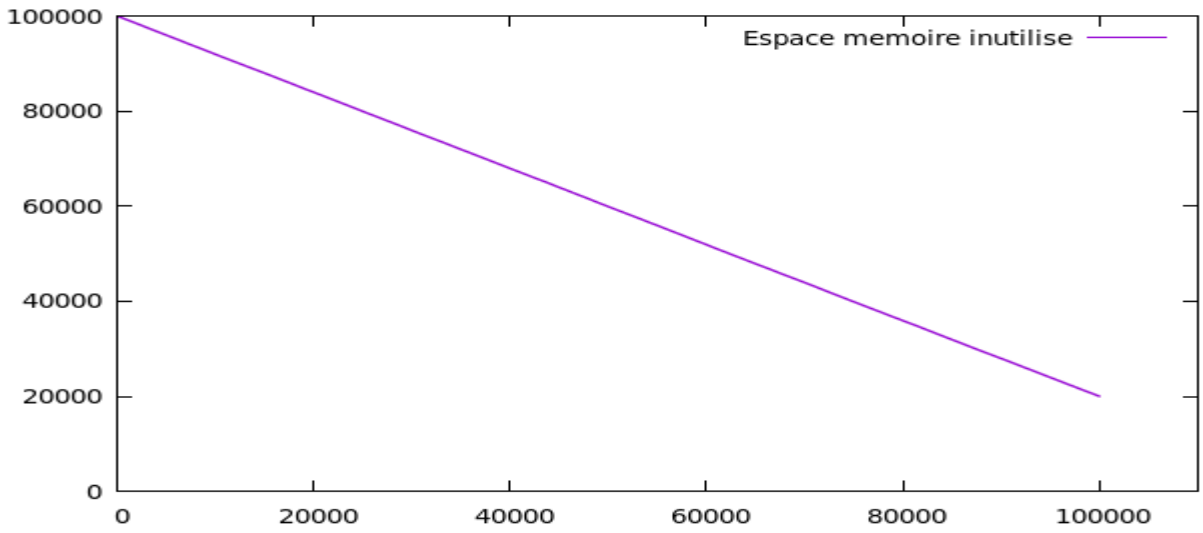
p=0.7


Amortized Java
100000
80000
60000
40000
20000
0
0
20000
40000
60000
80000
100000

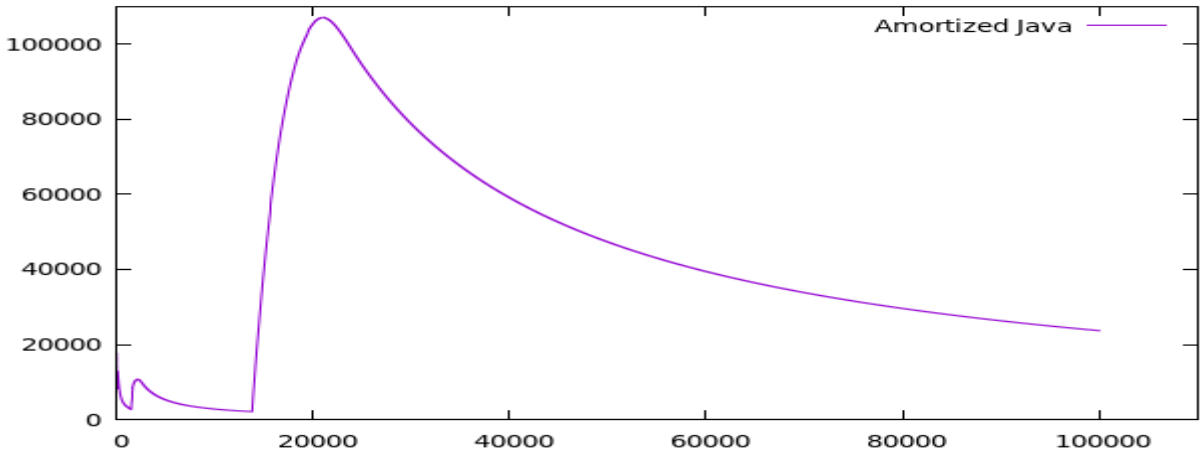

Espace memoire inutilise
100000
80000
60000
40000
20000
0
0
20000
40000
60000
80000
100000

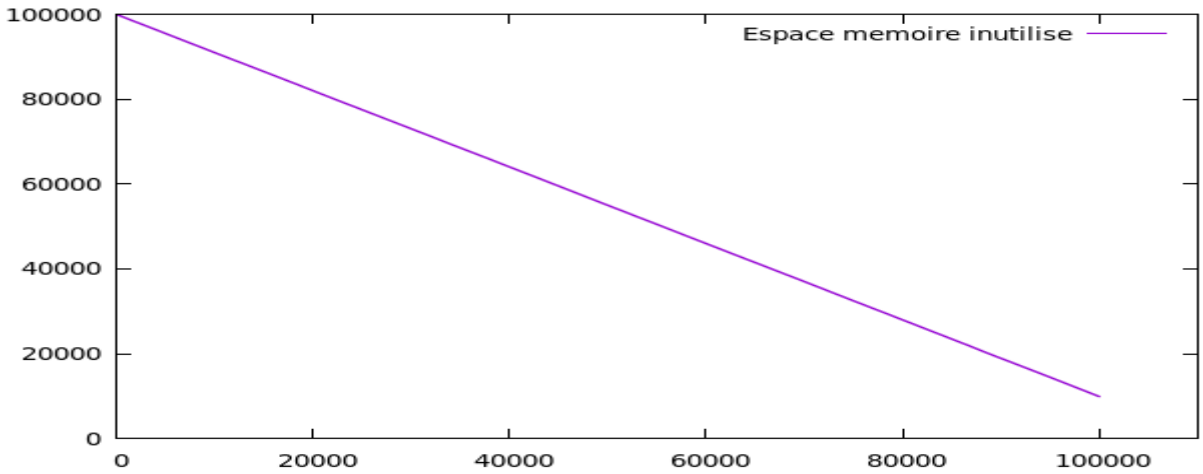
p=0.8


Amortized Java
80000
70000
60000
50000
40000
30000
20000
10000
0
0
20000
40000
60000
80000
100000


Espace memoire inutilise
100000
80000
60000
40000
20000
0
0
20000
40000
60000
80000
100000

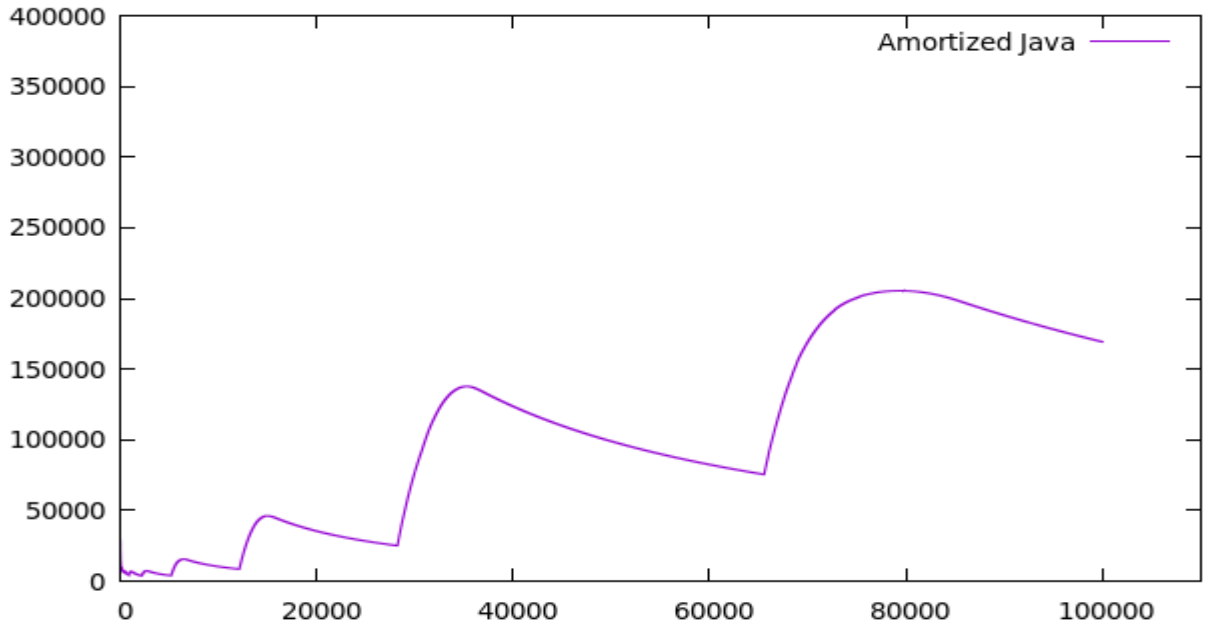
p=0.9

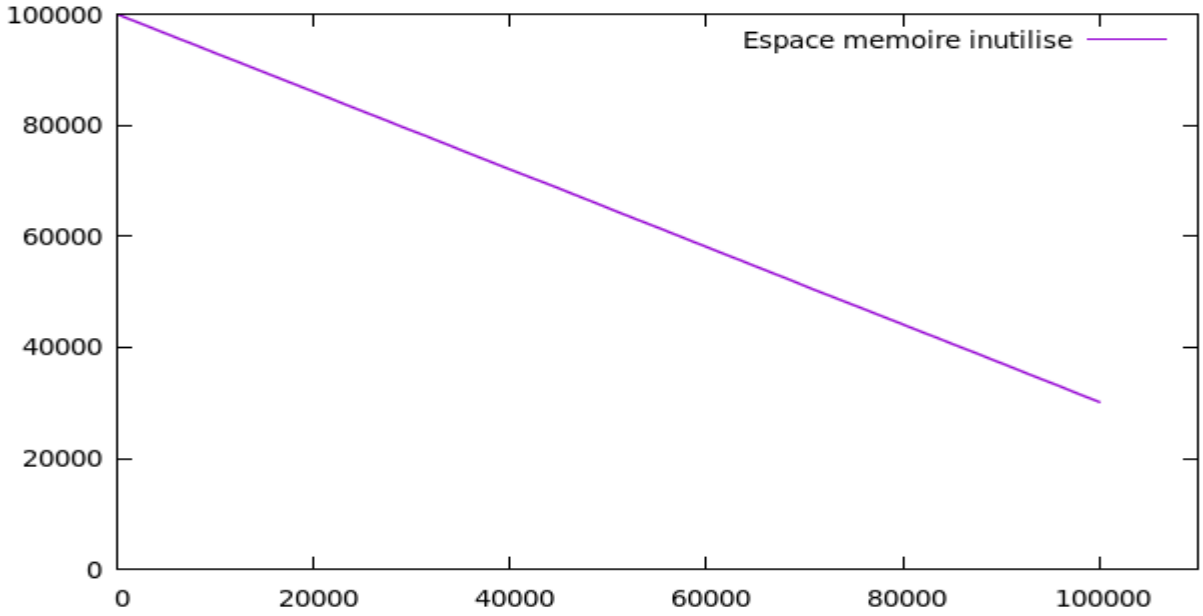




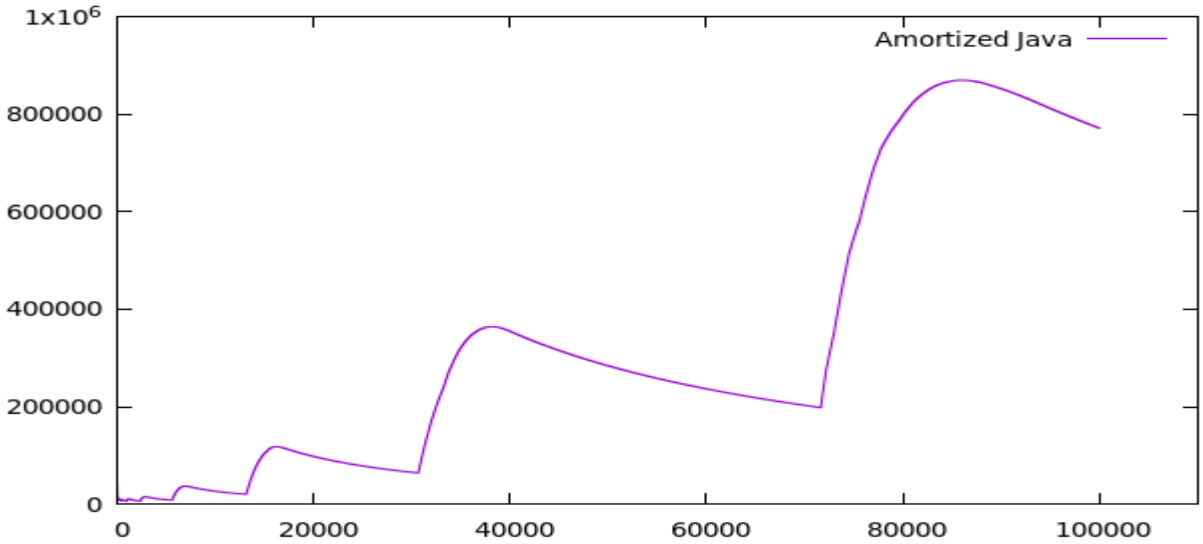
Ajouts et extractions des clés dans plusieurs tas

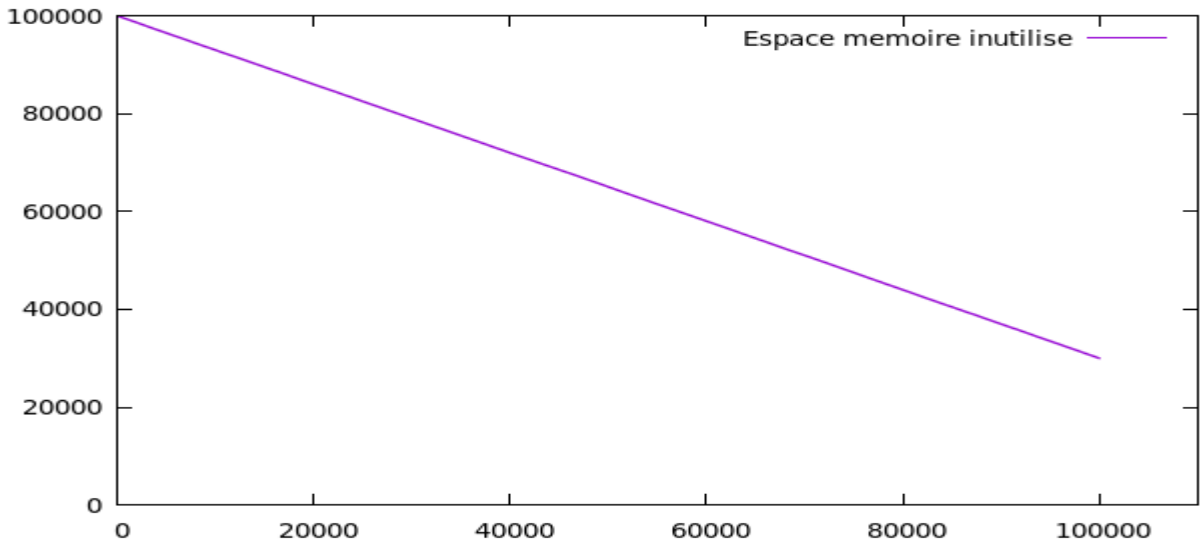
- ***On ajoute et extrait des clés dans 2 tas***





◆ ***On ajout et extrait les clés dans 4 tas***





Analyse

En ce qui concerne l'ajout des clés dans un tas binomial, nous l'avons expérimenté dans 3 cas :

- Dans le cas où les clés sont ajoutés par ordre croissant, le coût amorti est élevé en début et baisse pour les opérations ultérieures, le gaspillage de mémoire est négligeable voire nul.
- Dans le cas où les clés sont ajoutés par ordre décroissant, le coût amorti est élevé en début et baisse pour les opérations ultérieures, le gaspillage de mémoire est négligeable voire nul.
- Dans le cas où les clés sont ajoutés aléatoirement, le coût amorti est élevé en début et baisse pour les opérations ultérieures, le gaspillage de mémoire est négligeable.
- Dans le cas où il y a à la fois l'ajout et l'extraction des clés, nous avons faits varier le taux d'insertions et d'extractions selon un paramètre p.
  - Quand p>0.5, nous constatons une augmentation important du coût amorti. En effet, le coût amorti est faible en début et augmente au fur et à mesure qu'on effectue davantage des opérations dans le tas. L'espace mémoire inutilisée est aussi très grande.
  - Quand p<=0.5, nous n'avons pas pu effectuer d'expériences.
- Quand on insère et on extrait les clés dans plusieurs tas à la fois, le coût amorti augmente encore plus. Quant à l'espace mémoire inutilisée, il est .

Synthèse

Quand on effectue uniquement des insertions dans un tas binomial, le coût amorti est faible et on ne gaspille pas beaucoup de mémoire. En revanche, quand on effectue à la fois des insertions et des extractions dans un tas binomial, le coût amorti est très élevé, il en va de même pour le gaspillage de mémoire.
Ce qui nous permets d'en déduire, que l'opération d'ajout dans un tas binomial est moins coûteux. Par contre quand on fait à la fois des insertions et d'extractions il est très coûteux.

TP4 : B-ARBRES ET AVL

2) **Explication brève des choix d'implémentation**

Les programmes implémentant les B-Arbres et AVL sont écrits en Java.

La classe implémentant les B-Arbres contient les attributs et fonctions de bases suivantes :

- ***racine*** : attribut de type Noeud à partir de laquelle sont effectuées toutes les opérations ci-dessous
- ***capacity()*** : retourne la taille fixe de la structure. Cette taille est initialisée à la création de l'objet instanciant cette classe
- ***size()*** : retourne la taille dynamique de la structure. Cette taille varie selon qu'on fait l'insertion ou l'extraction des clés dans la structure. Cette taille est initialisée à 0 dans le constructeur
- ***parcoursArbre()*** : parcourt tous les nœuds de l'arbre
- ***rechercheCleArbre(int k)*** : recherche un nœud dont la clé est k
- ***insertionCleArbre(int k)*** : insère une clé dans un nœud donné et met à jour l'arbre
- ***supprimerCleArbre(int k)*** : extrait, supprime et met à jour l'arbre
- ***Noeud*** : classe métier à partir de laquelle sont implémenter toutes les opérations sur les nœuds

La classe implémentant les AVL contient les attributs et fonctions de bases suivantes :

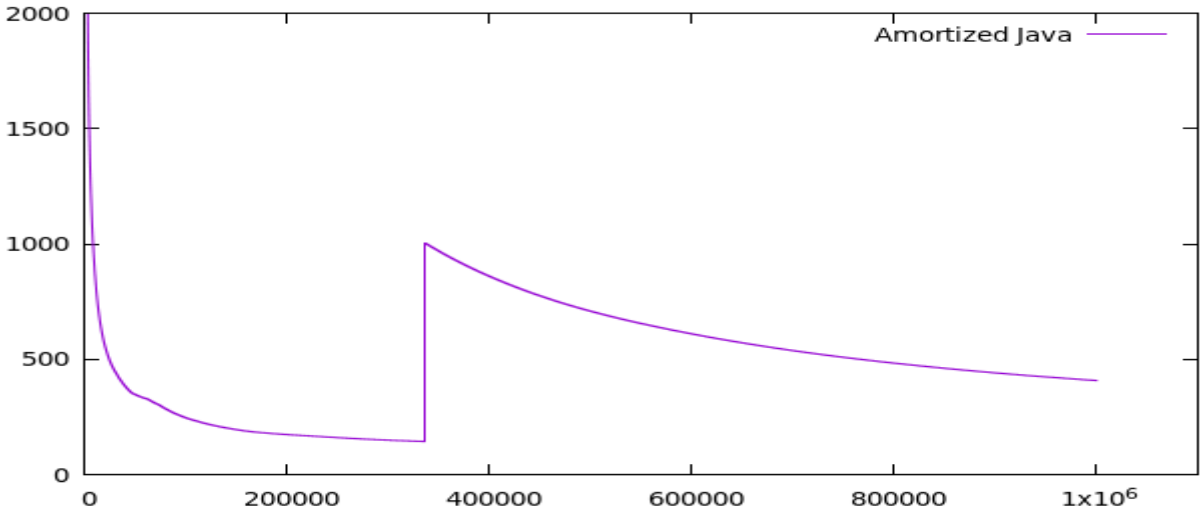
- ***root*** de type Node, à partir de laquelle sont effectuées toutes les opérations suivantes
- ***getSize()*** : retourne la taille courante de la structure
- ***getCapacity()*** : retourne la capacité de la structure
- ***insert(Node node, int key)*** : insère une clé à une position donnée du nœud
- ***deleteNode(Node node, int key***) : supprime une clé à une position donnée du nœud

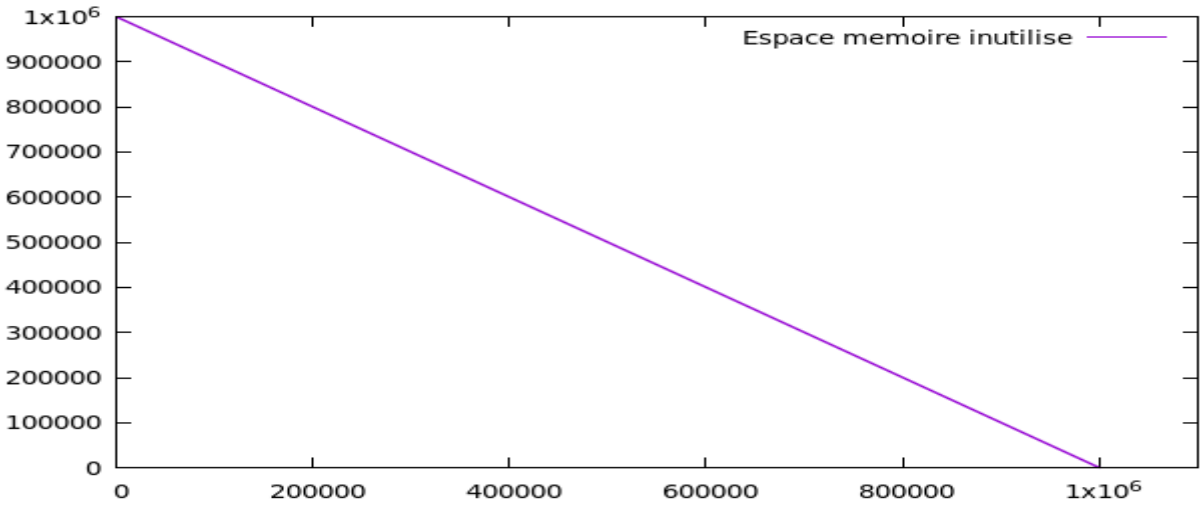
A la création de la structure, la capacité est fixée à 10^6 et la taille courante est initialisée à 0.

4) Expériences sur les B-Arbres et AVL

Expériences sur les B-Arbres

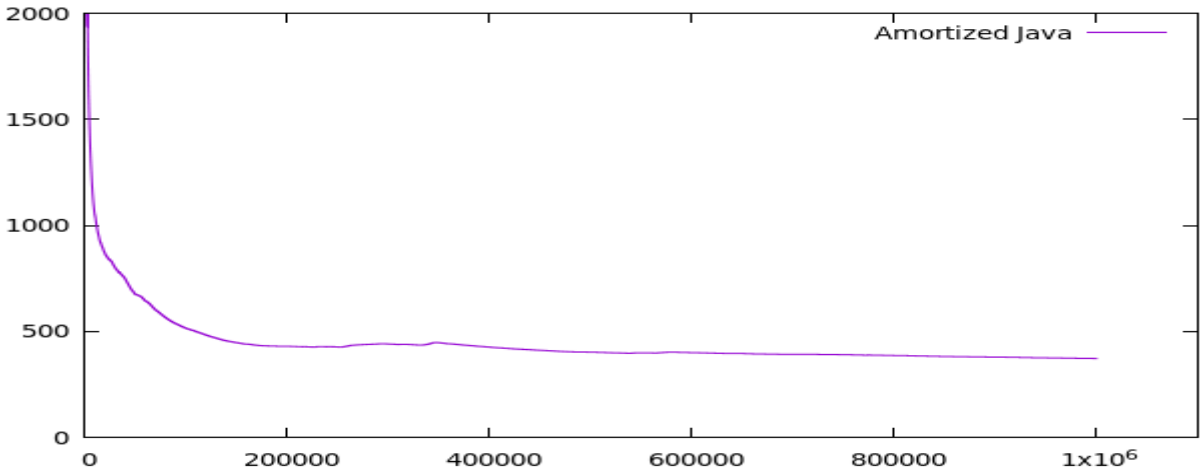
- **Les clés sont ajoutées dans l'ordre croissant**

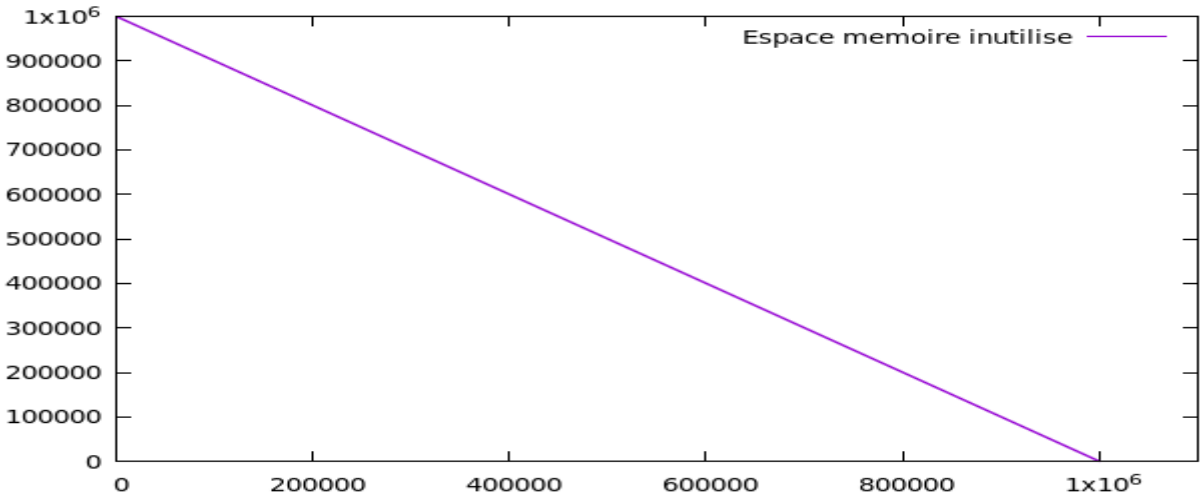




Analyse :
Quand les clés sont ajoutés dans l'ordre croissant, le coût amorti est élevé en début et pour les opérations ultérieures. L'espace mémoire inutilisée est nul.

- **Les clés sont ajoutées de façon aléatoire**

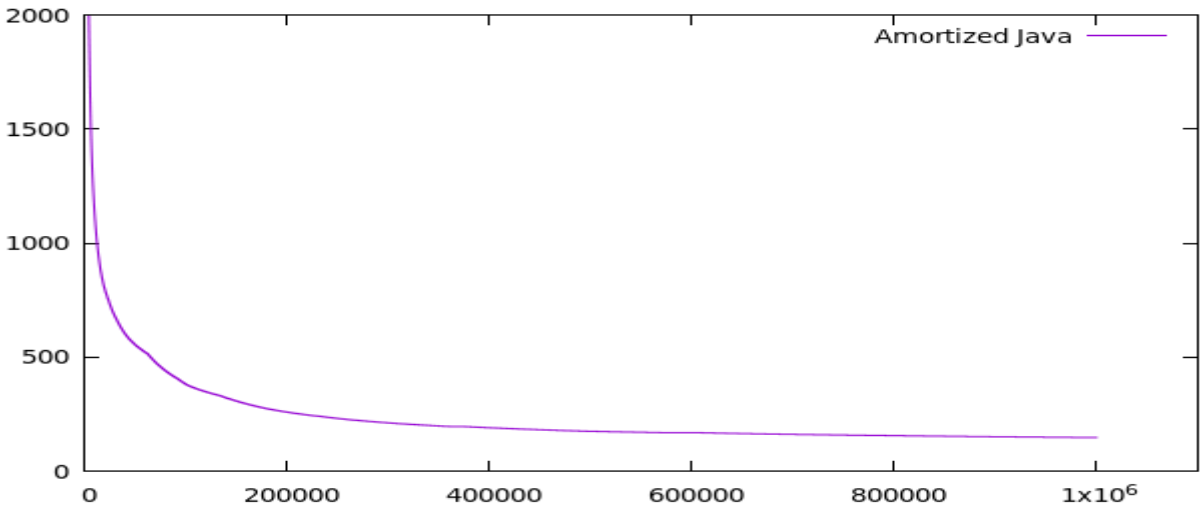


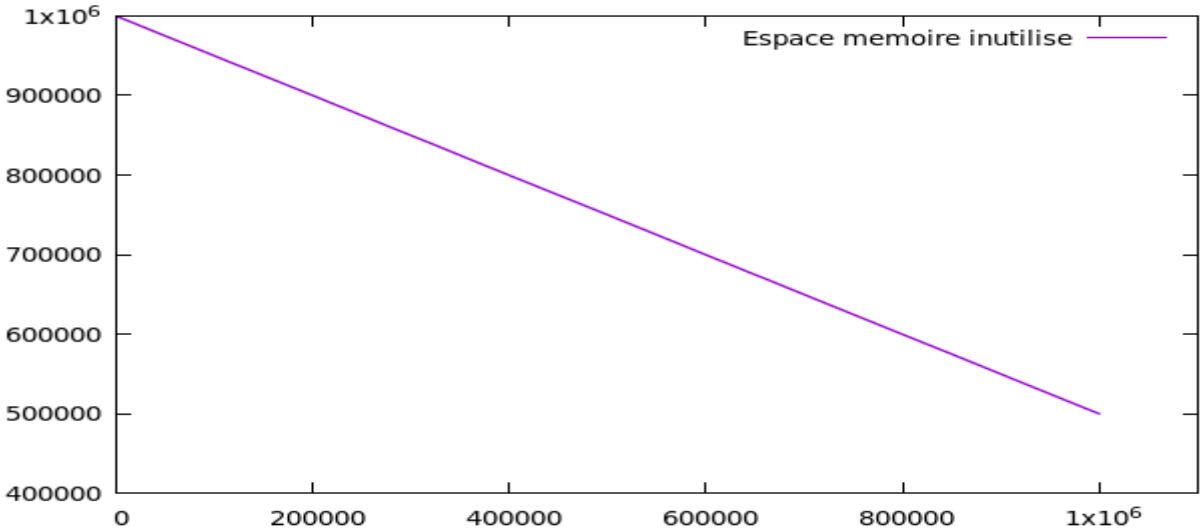


Analyse :

Quand les clés sont ajoutées de façon aléatoire, le coût amorti est élevé au début et pour les opérations ultérieures ; il est plus élevé que lorsque les clés sont ajoutées par ordre croissant. L'espace mémoire inutilisée est nul.

- **On ajoute et on extrait des clés**



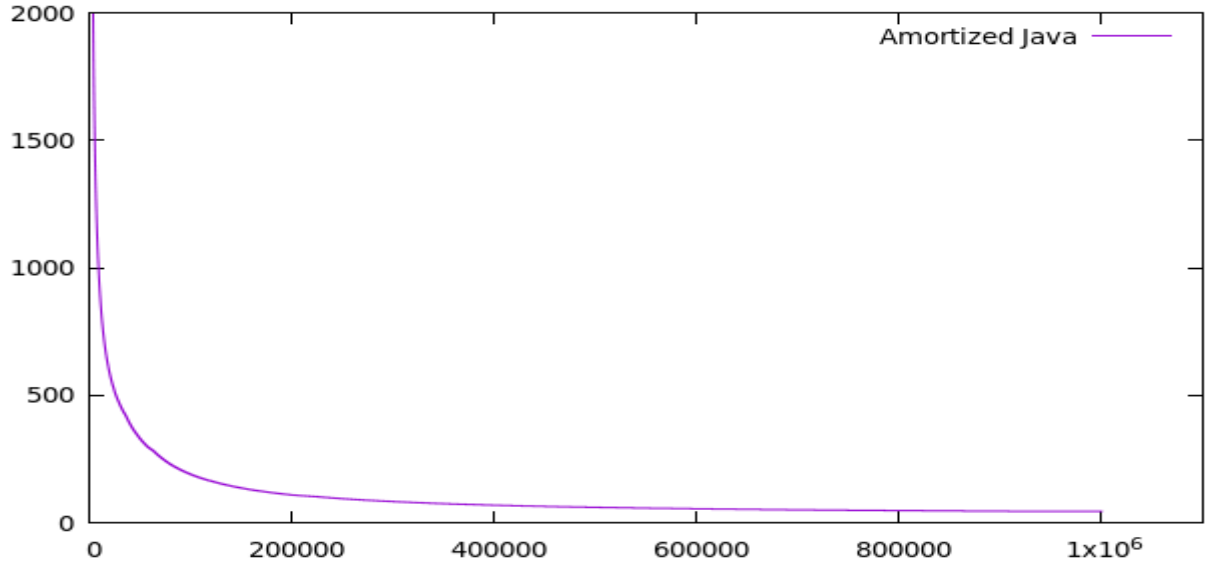


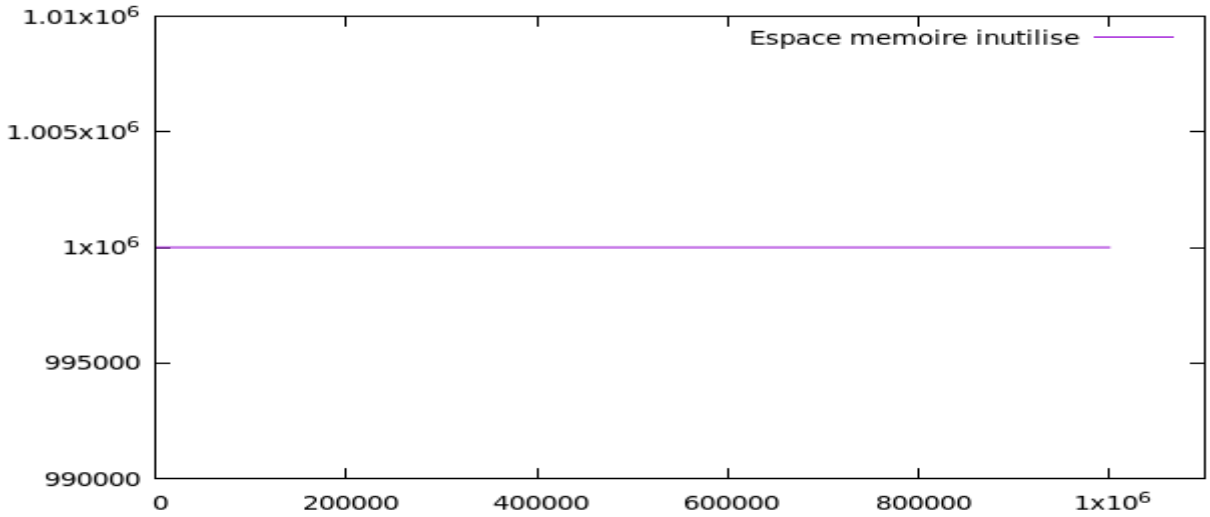
Analyse

Quand on effectue à la fois l'ajout et l'extraction des clés, le coût amorti reste élevé au début et pour les opérations ultérieures mais moins élevé que dans le cas où les clés sont ajoutées aléatoirement. En ce qui concerne l'espace mémoire inutilisée est élevé, soit près de la moitié des opérations requises dans la structure.

Expériences sur les AVL

- **Les clés sont ajoutées dans l'ordre croissant**

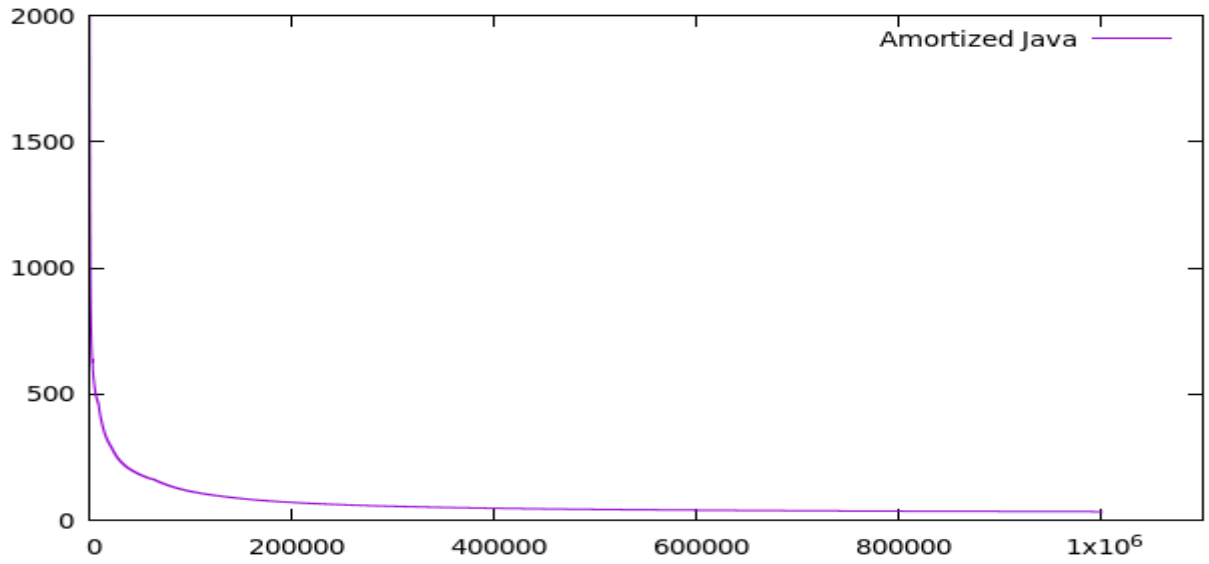


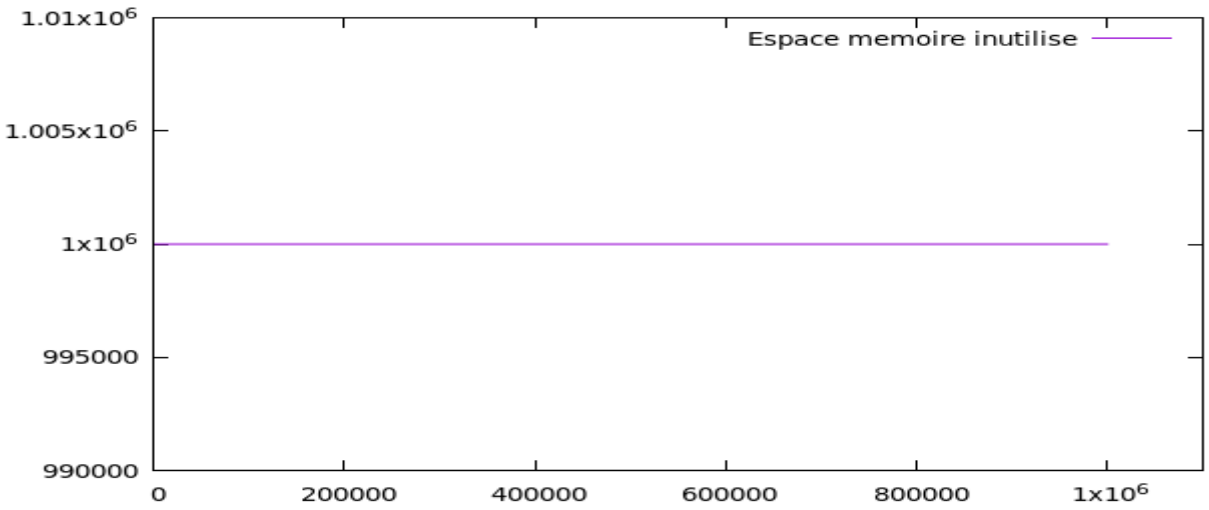


Analyse

Quand les clés sont ajoutés dans l'ordre croissant, le coût amorti est élevé au début et diminue pour les opérations ultérieures. L'espace mémoire inutilisée est constant tout au long des opérations. Le gaspillage de mémoire est très élevé, soit le nombre d'opérations nécessaires.

- **Les clés sont ajoutées de façon aléatoire**

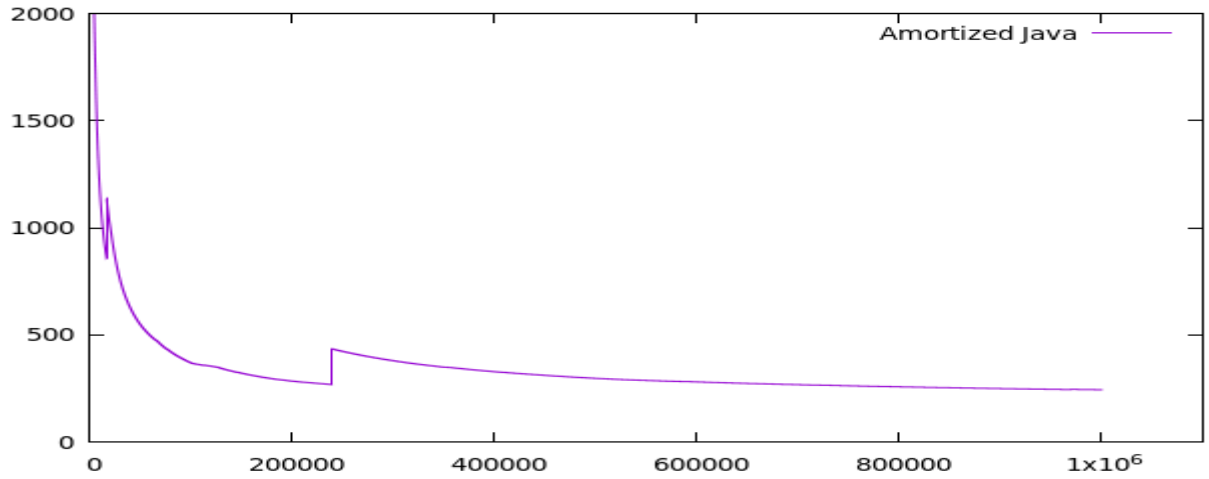


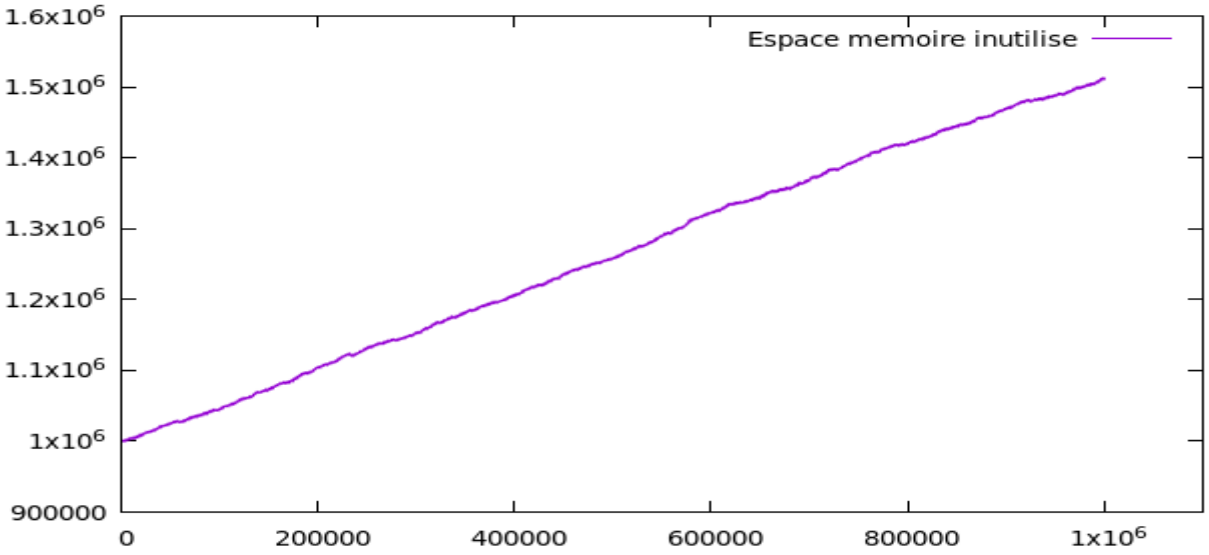


Analyse

Quand les clés sont ajoutées de façon aléatoire, le coût amorti est élevé au début et diminue encore pour les opérations ultérieures. L'espace mémoire inutilisée est constant tout au long des opérations. Le gaspillage de mémoire est important.

- **On ajoute et on extrait des clés**





Analyse

Quand on effectue à la fois l'ajout et l'extraction des clés dans un arbre AVL, le coût amorti est élevé au début et diminue pour les opérations ultérieures mais il est plus élevé que dans les 2 cas précédent. De constant, dans les cas précédents, l'espace mémoire inutilisée est très grand.

Synthèse:

L'analyse des expériences sur les B-arbres et arbres AVL nous permettent d'en déduire que :

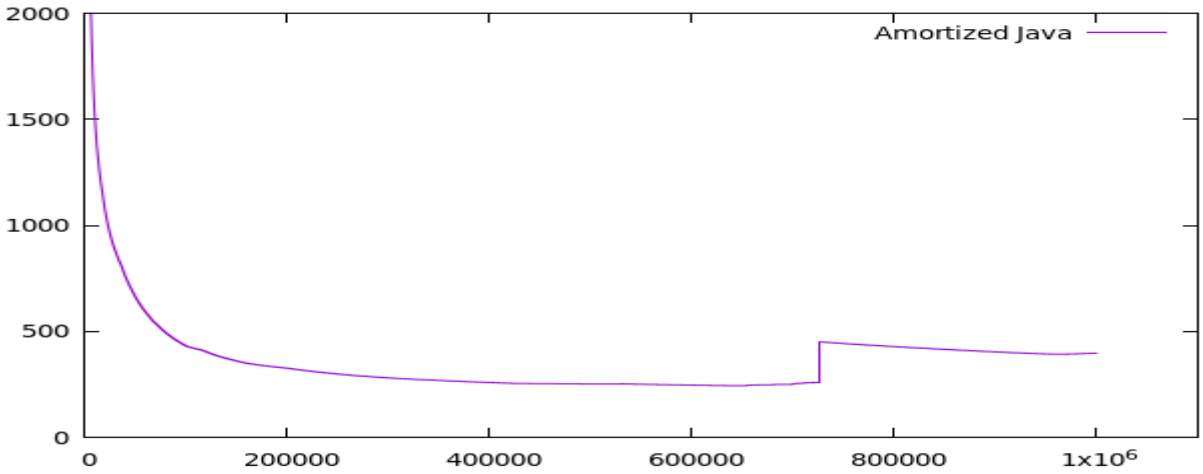
- Lorsque les clés sont ajoutées dans l'ordre croissant ou de façon aléatoire, le coût amorti est plus faible dans les AVL par rapport aux B-Arbres. En revanche, le gaspillage de mémoire est plus important dans les AVL par rapport aux B-Arbres.
- Lorsqu'on effectue à la fois l'ajout et l'extraction des clés, le coût amorti est élevé dans les AVL par rapport aux B-Arbres. En revanche, l'espace mémoire inutilisée est équivalent dans les 2 structures.

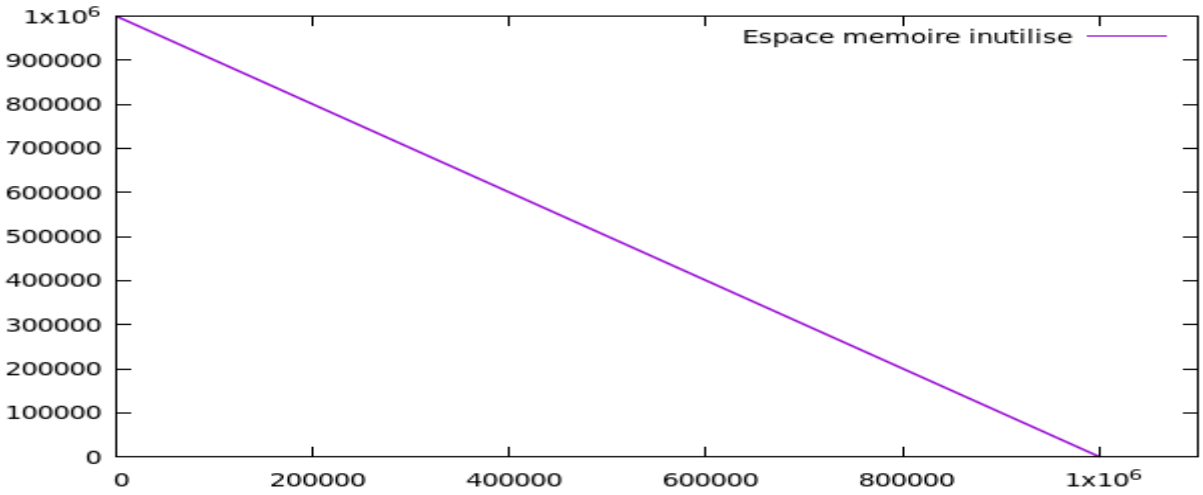
Dans l’expérience ci-dessus, on obtient un meilleur résultat dans les 2 structures quand les clés sont ajoutées dans l’ordre croissant.
Dans le cas où les clés sont ajoutées de façon aléatoire, il est difficile pour nous de nous prononcer sur l’efficacité ou pas d’une structure de données car on ne peut savoir au préalable l’ordre dans lequel les clés seront tirées.
Au regard de toutes ces analyses, on peut dire qu’on obtient un meilleur temps d’exécution dans les AVL par rapport aux B-Arbres ; d’autres part on gaspille moins de mémoire dans les B-Arbres par rapport aux AVL.

5) Étude de l’efficacité en fonction du degré minimum

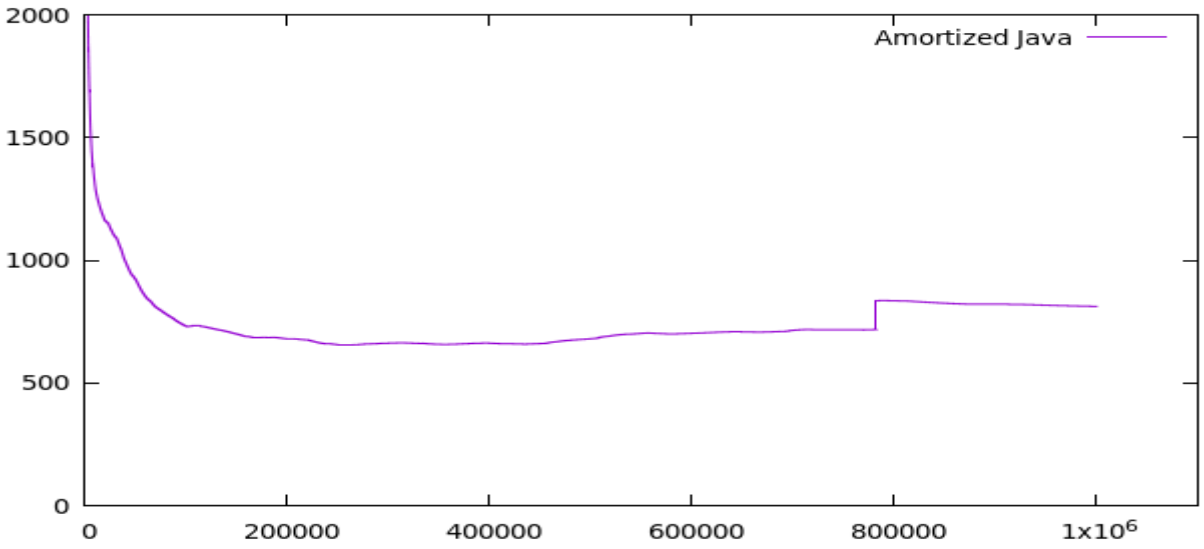
Pour t =2

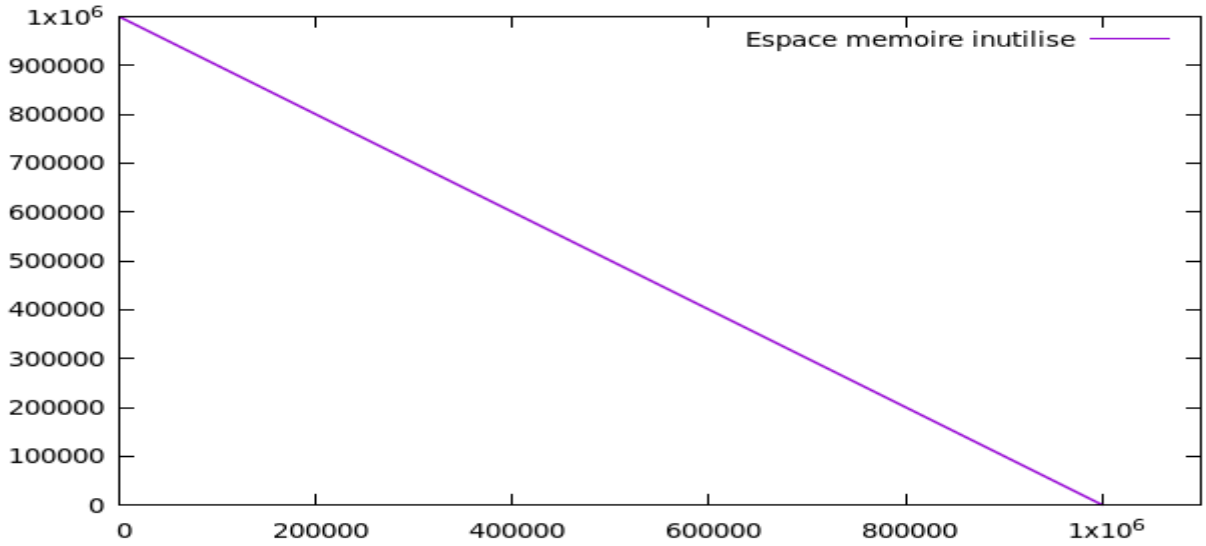
- **Clés ajoutées dans l’ordre croissant**



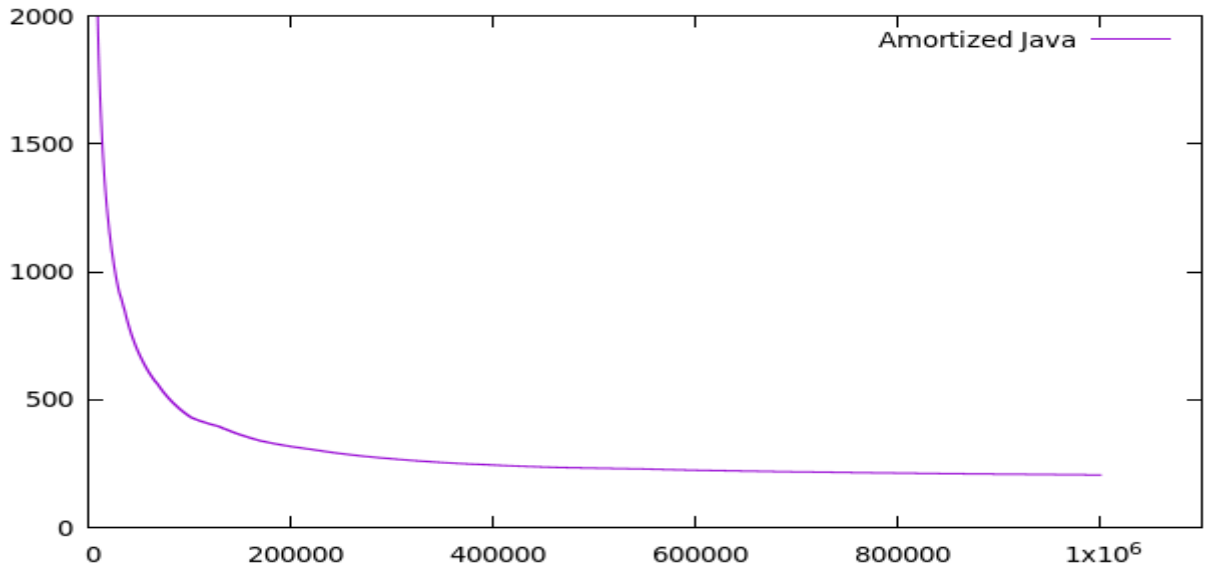


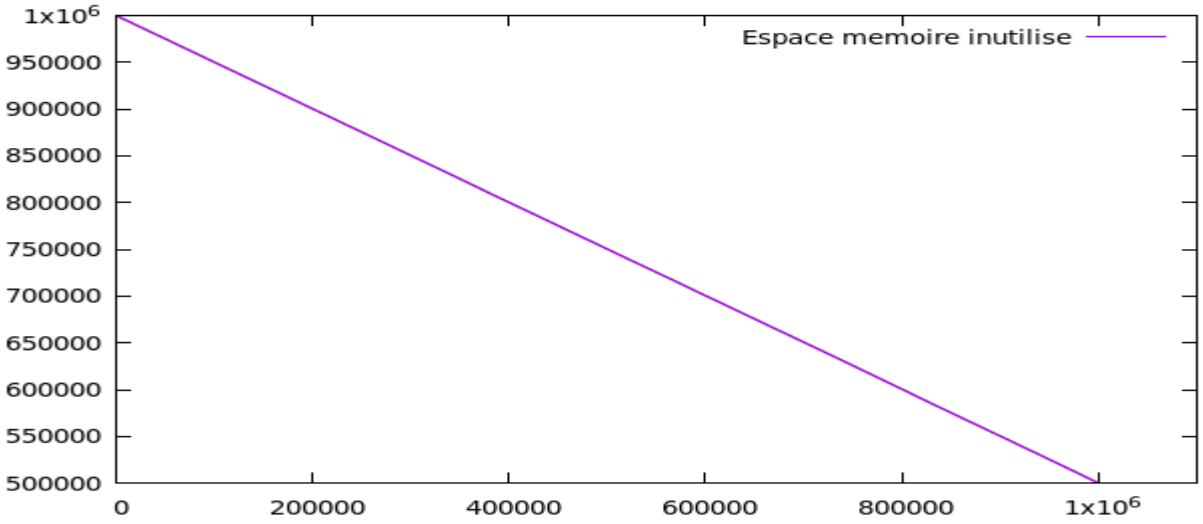
■ **Clés ajoutées aléatoirement**





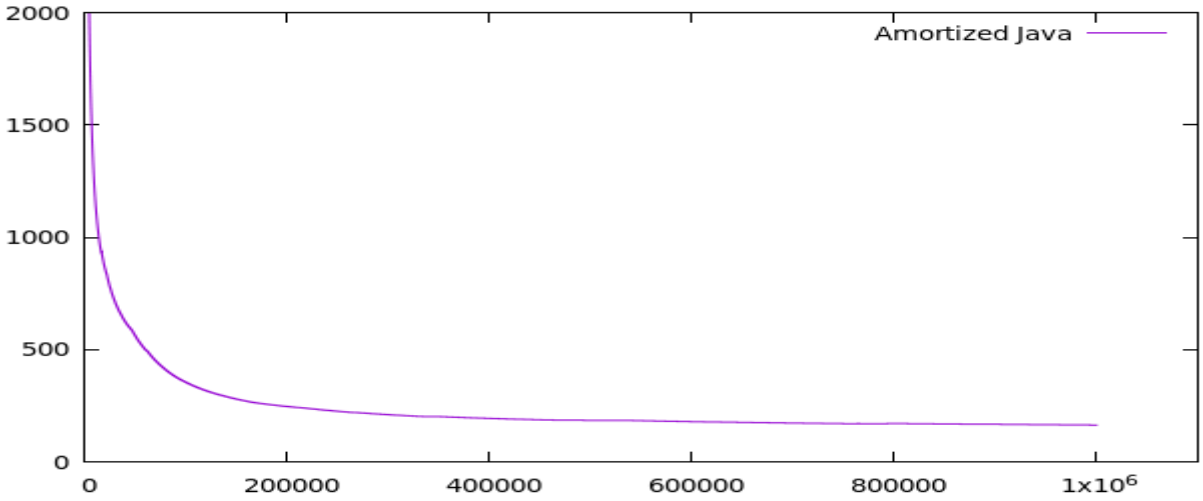
- **Ajout et Extraction des clés**

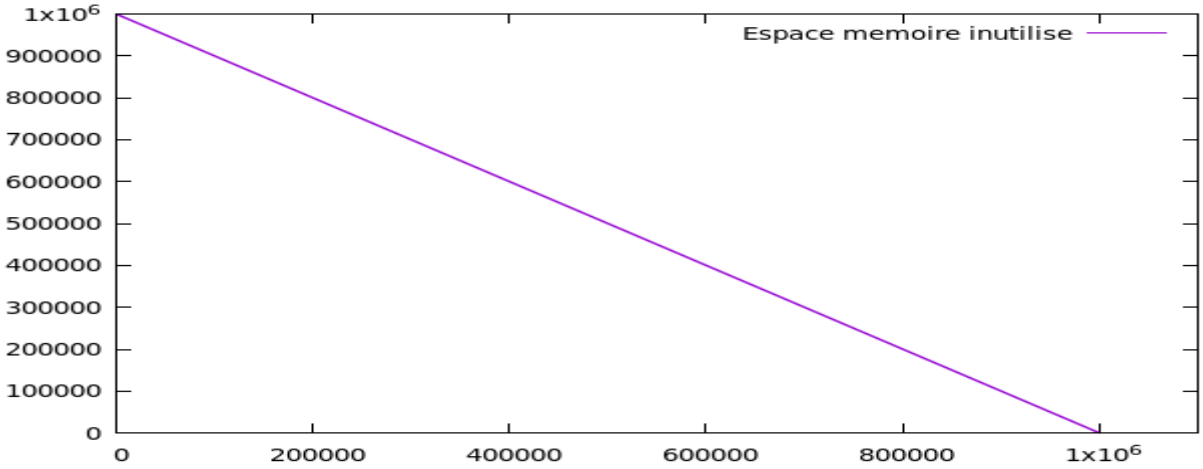




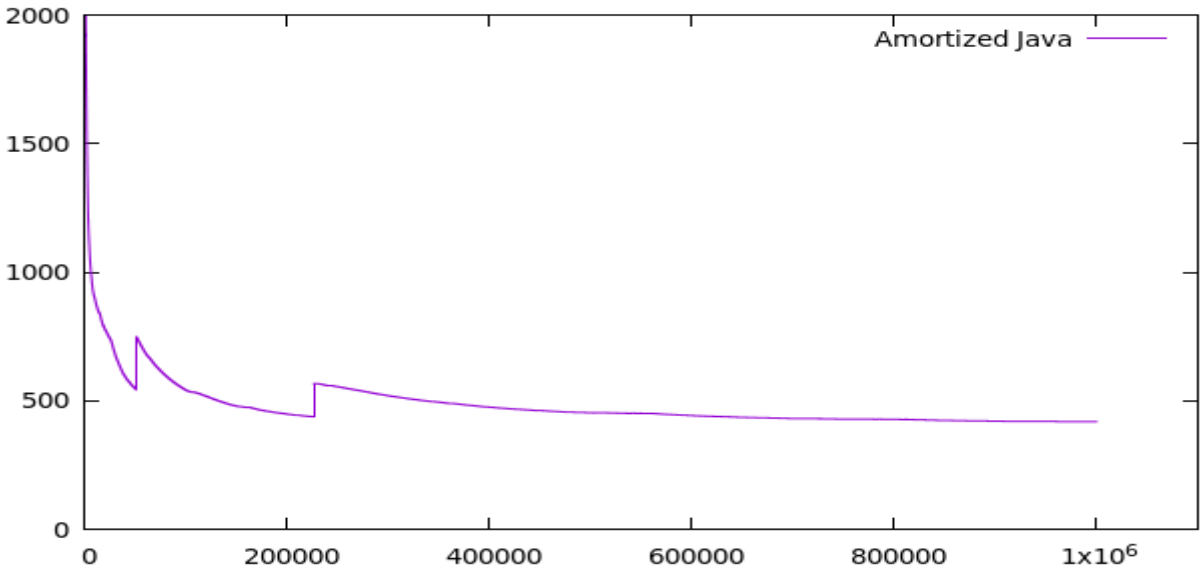
Pour t=3

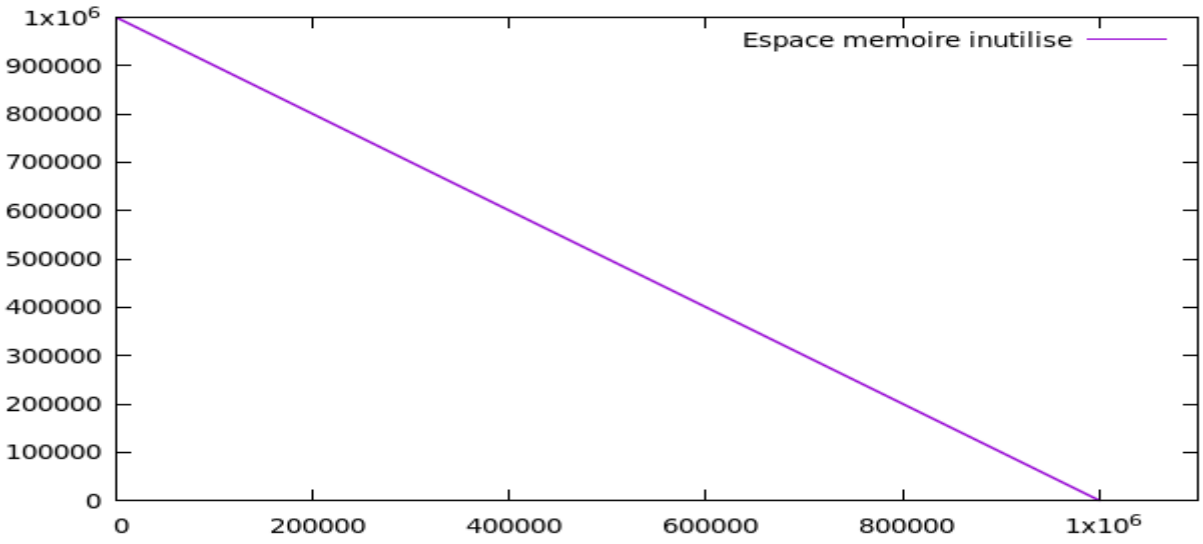
- **Clés ajoutées dans l'ordre croissant**



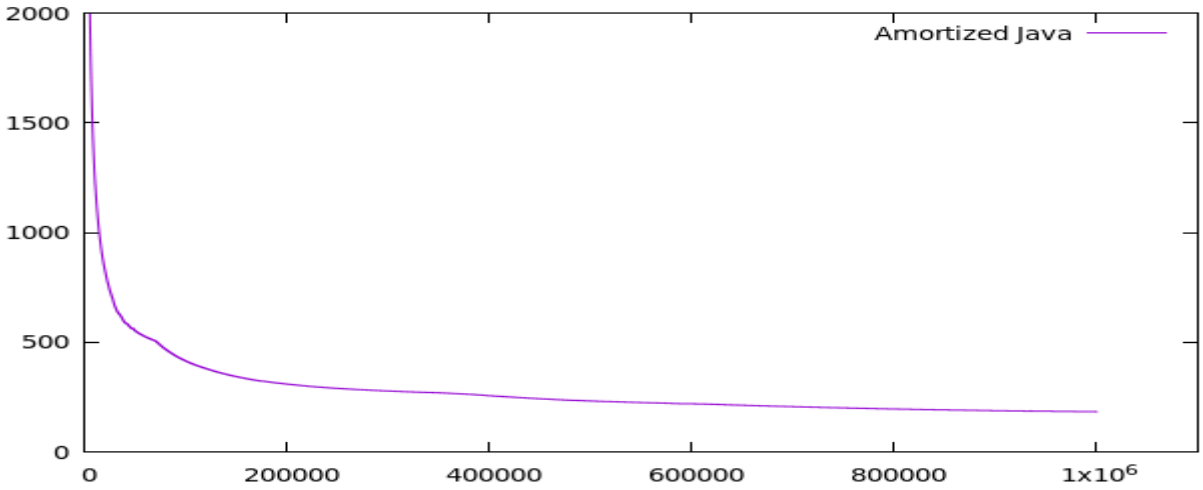


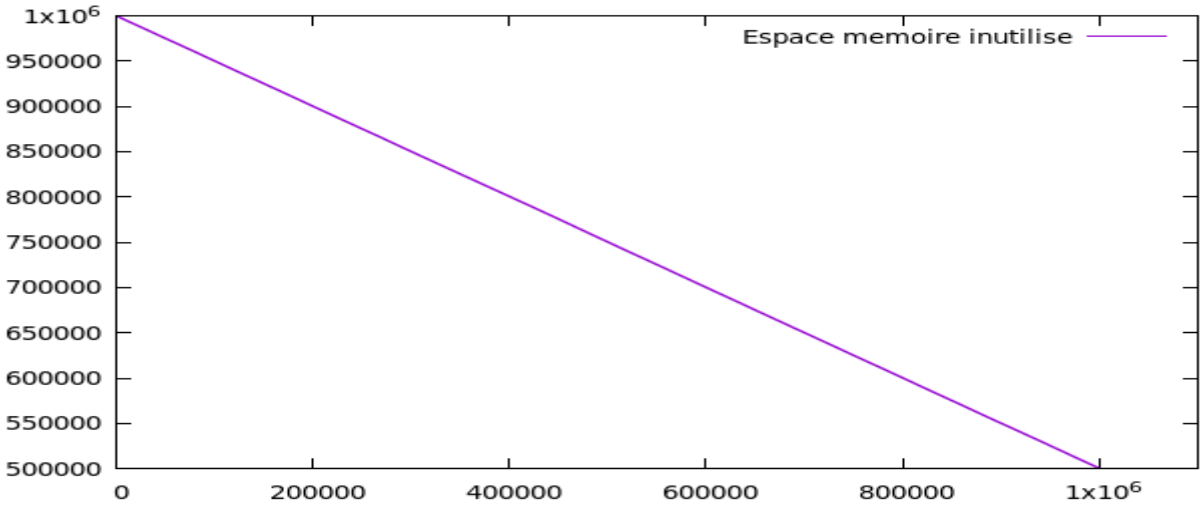
- **Clés ajoutées aléatoirement**





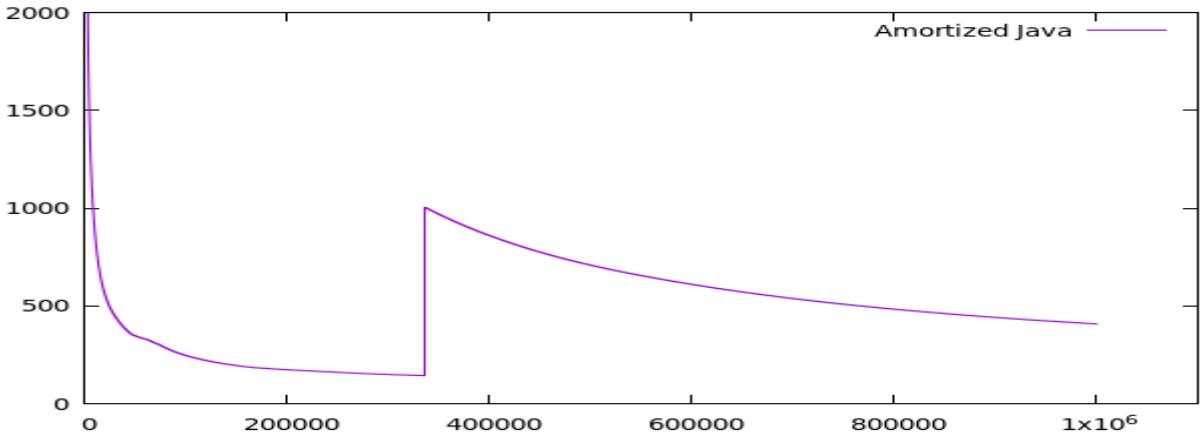
■ **Ajout et Extraction des clés**

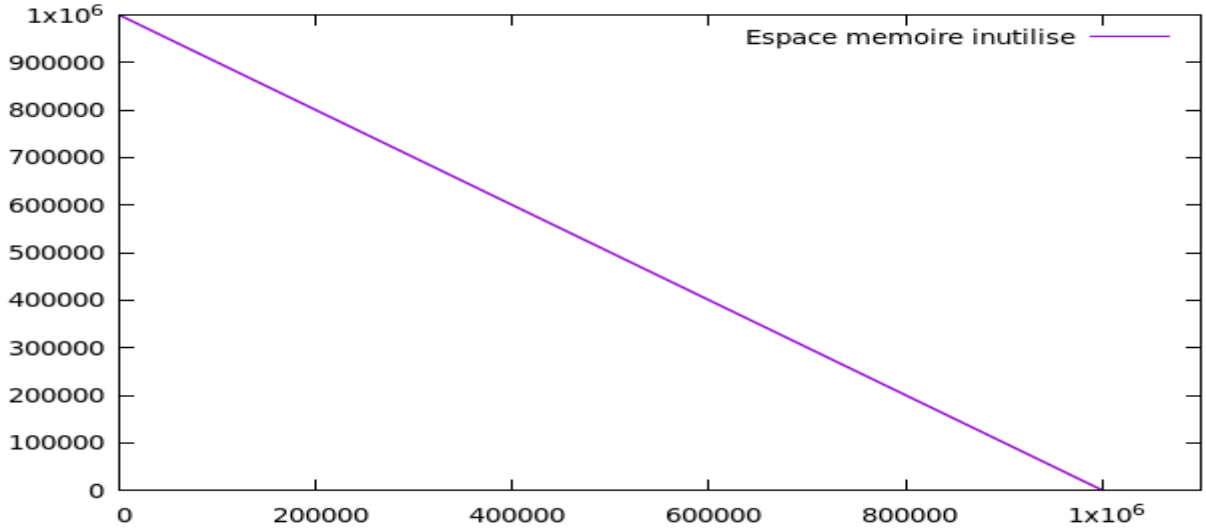




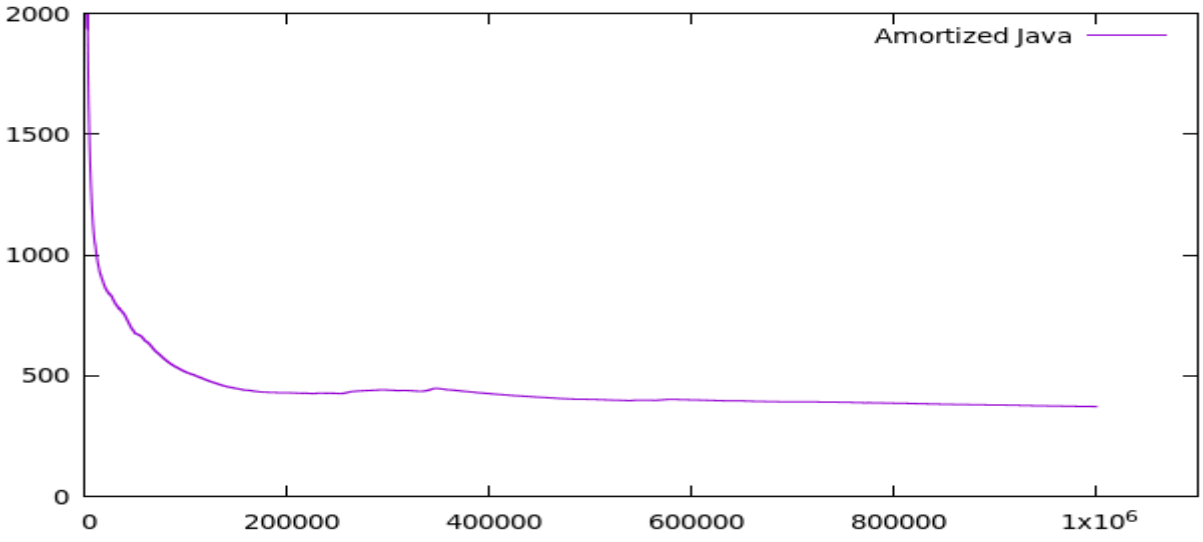
Pour t= 4

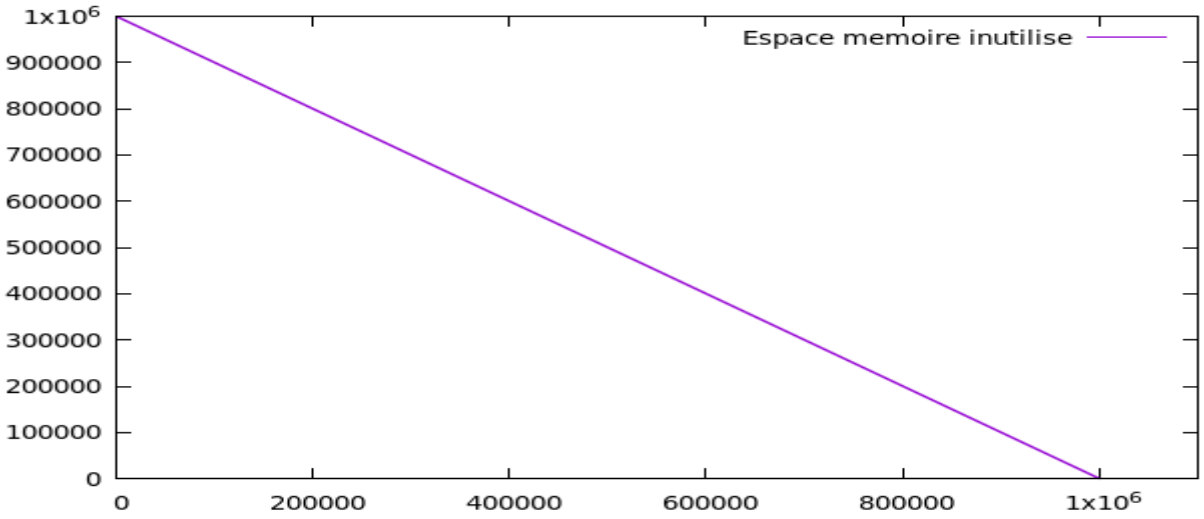
- **Clés ajoutées dans l'ordre croissant**



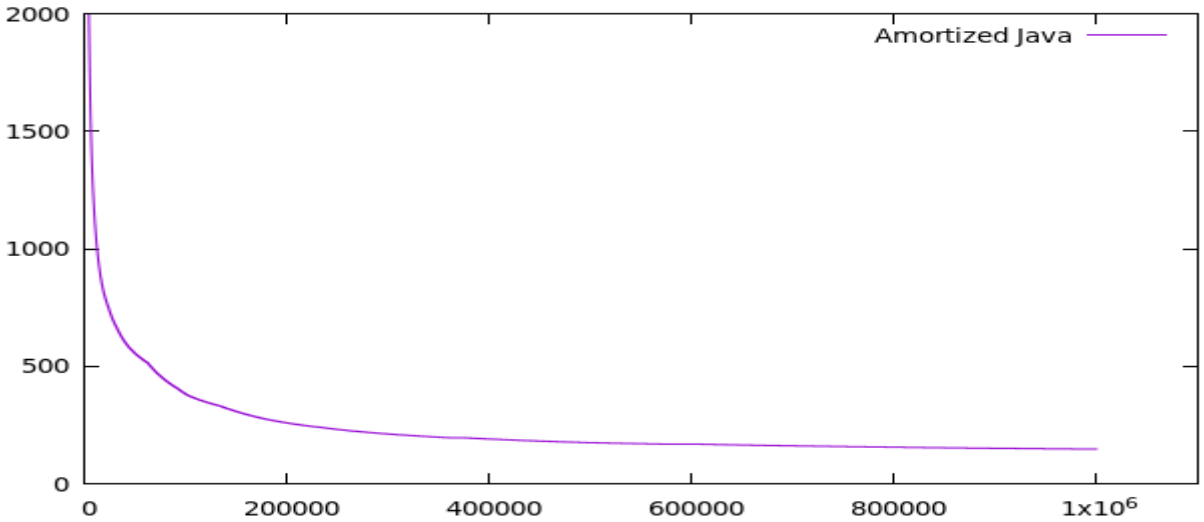


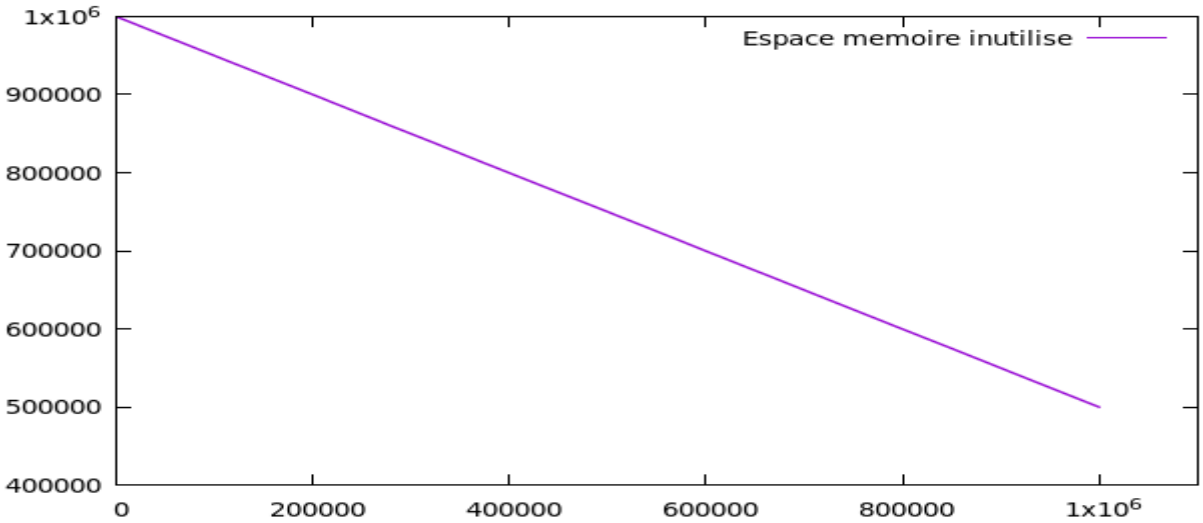
- **Clés ajoutés aléatoirement**





■ **Ajout et Extraction**





Analyse

En faisant varier le degré minimal t, nous constatons qu'on a des meilleurs résultats quand t est grand. Selon qu'on ajoute uniquement les clés (croissantes ou aléatoires) ou on ajoute et on extrait les clés, on obtient le même espace inutilisé selon que t=2, t=3 et t=4. En revanche, le coût amorti change selon les valeurs de t.

Quand on ajoute les clés dans l'ordre croissant, le coût amorti est plus faible pour t=3, ensuite t=4 et enfin t=2.

Quand on ajoute les clés aléatoirement, le coût amorti est plus faible pour t=4 puis t=3 et enfin t=2.

Quand on ajoute et on extrait les clés, le coût amorti est plus faible pour t=4 puis t=3 et enfin t=2.

Synthèse

Au regard de ces analyses, on peut dire que plus de degré minimal est grand, plus on obtient un meilleur temps d'exécution du programme. Pour les valeurs de t expérimentées, on a des meilleurs résultats pour t=4 ;

Conclusion

A travers ces TPS, nous avons fait une analyse des coûts en temps et en mémoire des opérations dans les tables dynamiques, les tas binaires, les tas binomiaux, les B-arbres et les AVL.